Thesis for the Degree of Doctor of Philosophy

Verification of Distributed Erlang Programs using Testing, Model Checking and Theorem Proving

Hans Svensson

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Göteborg
Sweden

Göteborg, 2008

Verification of Distributed Erlang Programs using
Testing, Model Checking and Theorem Proving

Hans Svensson
ISBN 978-91-7385-096-4

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 2777
ISSN 0346-718X

Technical Report no. 38D
Department of Computer Science and Engineering
Division: Software Engineering and Technology
Research Group: Functional Programming

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Göteborg, Sweden
Telephone +46 (0)31–772 1000

Printed at the Department of Computer Science and Engineering
Göteborg, Sweden 2008

Abstract

Software infiltrates every aspect of modern society. Production, transportation, entertainment, and almost every other sphere that influences modern living are either directly or indirectly dependent on software systems. Software systems provide such a degree of flexibility that their role as a driving force for new and better products is indisputable.

The downside is that software systems are rarely error-free. For a plentitude of reasons most software systems contain errors. Software errors impose large costs; the more important the system, the higher is the cost of an error. Reports show that a normal software development project spends 40% to 50% of its time and budget on quality assurance. Thus, software project economy is a great incitament for research of better tools and methods for software development.

This thesis is part of the continuous efforts of finding more efficient software development methods and addresses the problem of *verifying algorithm implementations*. We have in particular studied algorithms designed for *distributed* (systems composed of a collection of computers, processors or processes) and *fault-tolerant* systems (systems designed to withstand some degree of failure). Verification of distributed and fault-tolerant systems is notoriously hard because both the distribution and the fault-tolerance add complexity to the software systems. The thesis introduces, motivates and evaluates several different verification methods related to distributed and fault-tolerant algorithm implementations.

We introduce a *trace-based* testing method, which has been used to find and analyze errors in an existing open-source implementation of a fault-tolerant leader election algorithm. In the thesis we also present a new open-source implementation of a leader election algorithm, which is based on a verified algorithm by Stoller.

We have developed a distributed semantics for Erlang. Errors found using trace-based testing indicated that existing Erlang semantics were not detailed enough. We propose a fully distributed extension of an existing single-node semantics for Erlang.

We present McErlang, an explicit state model checker implemented in Erlang, and using Erlang as its specification language. We demonstrate that the 'all in Erlang'-approach to model checking is promising.

We propose a semi-automatic algorithm verification method that has been used to prove safety properties for Stoller's leader election algorithm. The verification method uses automated theorem provers to inductively prove first-order logic invariants.

This thesis is based on the work contained in the following papers:

1. *Semi-Formal Development of a Fault-Tolerant Leader Election Protocol in Erlang*, T. Arts, K. Claessen, and H. Svensson, In Lecture Notes in Computer Science, vol. 3395, p. 140-153, Springer, Feb 2005.
2. *A New Leader Election Implementation*, H. Svensson and T. Arts, In Proc. of the ACM SIGPLAN 2005 Erlang Workshop, Tallinn, 2005.
3. *A More Accurate Semantics for Distributed Erlang*, H. Svensson and L-Å. Fredlund, In Proc. of the ACM SIGPLAN 2007 Erlang Workshop, Freiburg, 2007.
4. *McErlang: A Model Checker for a Distributed Functional Programming Language*, L-Å. Fredlund and H. Svensson, In Proc. of International Conference on Functional Programming (ICFP), ACM SIGPLAN, Freiburg, 2007.
5. *A Semi-Automatic Correctness Proof Procedure applied to Stoller's Leader Election Algorithm*, H. Svensson, Technical Report no. 2008:7, Computer Science and Engineering, Chalmers University of Technology, 2008.
6. *Finding Counter Examples in Induction Proofs*, K. Claessen and H. Svensson, published at 'The Second International Conference on Tests and Proofs' (TAP), in Prato, Italy, April 2008.

My contributions to these papers are:

1. I implemented the testing framework, re-using parts of earlier work by Arts, and did all of the testing and trace analysis. All three authors were involved in the discussion and presentation of the results. I wrote most of section 2; apart from that the writing was equally divided among the authors.
2. My co-author did the first prototype implementation, which I then refined and optimized. I performed the first round of trace-based testing, and we were both involved in the QuickCheck testing. The writing was a joint effort, where I was involved in writing all sections.
3. My co-author designed the single node semantics for Erlang briefly described in section 2. I then designed and presented the distributed semantics for Erlang. The paper is based on an earlier presentation of the distributed semantics [CS05], written by me together with my supervisor. Both my supervisor and my co-author contributed with good ideas and design proposals. I wrote most of the paper with valuable additions by my co-author.
4. McErlang was designed and implemented by my co-author. I independently designed and implemented a similar prototype system in Haskell, and some of the ideas from there are also part of McErlang. My co-author did most of the writing of the main sections 4 and 5, while I concentrated on sections 3 and 6. The work on the other sections was equally divided among my co-author and me.
5. I implemented the proof framework, and constructed the needed invariants. The implementation and proof are the result of many (long) discussions with my supervisor. He was especially involved in the axiomatizations used in the proof procedure. I have written the paper myself, with many valuable comments from my supervisor.
6. The idea and problem statement comes mainly from my co-author. I did the prototype implementation presented in the paper. The writing was quite equally divided among my co-author and me. I wrote most of section 4 and was not very involved in writing section 1 of the paper.

Contents

Acknowledgements **xi**

Introduction **1**
1 Software Development . . . 2
1.1 Informal Software Verification . . . 2
1.2 Formal Verification . . . 3
2 Distributed Systems . . . 3
3 Fault-Tolerance . . . 4
4 Leader Election . . . 5
4.1 Ring Networks . . . 5
4.2 The Bully Algorithm . . . 6
4.3 Verification of Leader Election Algorithms . . . 7
5 Erlang . . . 8
5.1 Erlang Today . . . 9
5.2 Verification of Erlang Programs . . . 9
6 Implementing a Formally Verified Algorithm . . . 10
6.1 Semantic Incompatibility . . . 10
6.2 Context Adaptation . . . 11
6.3 Verifying the Implementation . . . 11
7 Abstractions . . . 14
8 Contributions . . . 16
9 Thesis overview . . . 17

Paper 1 – Semi-Formal Development of a Fault-Tolerant Leader Election Protocol in Erlang **29**
1 Introduction . . . 31
2 Methodology . . . 33
2.1 Fault-Tolerant Leader Election Version 1 . . . 33
2.2 Generating Stimuli and Tracing . . . 34
2.3 Abstractions . . . 35
2.4 Abstractions for Bug Finding . . . 36
2.5 Sanity Checks on Abstractions . . . 38
2.6 Fault-Tolerant Leader Election Version 2 . . . 39
2.7 Abstractions for Verification . . . 40
2.8 Coverage . . . 41

3 Related Work . . . 42
4 Conclusions and Future Work . . . 43

Paper 2 – A New Leader Election Implementation 47
1 Introduction . . . 49
2 Algorithm . . . 51
3 Implementation and Testing . . . 54
3.1 Testing with trace recording . . . 55
3.2 Testing with QuickCheck . . . 56
3.3 Coverage . . . 57
4 Discussion . . . 58
5 Acknowledgments . . . 59

Paper 3 – A More Accurate Semantics for Distributed Erlang 61
1 Introduction . . . 63
2 Original semantics . . . 65
3 Motivation . . . 69
3.1 Message reordering . . . 69
3.2 Disconnected nodes . . . 72
4 Distributed (Multi-Node) Semantics . . . 72
4.1 Nodes . . . 73
4.2 Node message queues . . . 73
4.3 Run-Time systems . . . 73
4.4 Changes to the single-node semantics . . . 74
4.5 Transitions . . . 75
4.6 Operational output rules . . . 76
4.7 Operational input rules . . . 76
4.8 Operational node rules . . . 77
4.9 Fairness . . . 79
4.10 Message reordering . . . 80
4.11 Node disconnection . . . 80
5 Properties of the Multi-Node Semantics . . . 81
5.1 Extension . . . 82
5.2 Message Reordering and Node Disconnect . . . 83
5.3 Expressiveness . . . 84
5.4 Finite systems stays finite . . . 84
5.5 A word of caution . . . 84
6 Discussion . . . 85
7 Related Work . . . 87
8 Conclusions and Future Work . . . 88

Paper 4 – McErlang: A Model Checker for a Distributed Functional Programming Language 91
1 Introduction . . . 94
2 The Erlang Programming Language . . . 96
3 Semantics . . . 97
3.1 World Hello? . . . 98

3.2 Semantics implemented in McErlang . . . 100
4 Structure of the Implementation . . . 100
4.1 Source Language . . . 100
4.2 Correctness Properties . . . 101
4.3 Algorithms . . . 102
4.4 Tables . . . 103
4.5 Abstractions . . . 104
5 Executing Erlang Programs in McErlang . . . 104
5.1 Run-time Organization . . . 105
5.2 Translation . . . 106
5.3 Data Structures in the Run-time System . . . 109
5.4 Model Checker Semantics . . . 110
5.5 Run-time Environment Modeling . . . 111
6 Evaluation . . . 113
6.1 Resource manager . . . 114
6.2 Leader election . . . 115
7 Discussion . . . 118
8 Related Work . . . 118
9 Conclusion and Future Work . . . 119

Paper 5 – A Semi-Automatic Correctness Proof Procedure applied to Stoller's Leader Election Algorithm 125
1 Introduction . . . 127
2 Background . . . 129
3 Algorithm . . . 130
3.1 General Description – The Bully Algorithm . . . 130
3.2 Garcia-Molina's Bully Algorithm . . . 131
3.3 Stoller's Bully Algorithm . . . 131
4 Proof Procedure . . . 132
4.1 Sanity Checks . . . 133
5 Model . . . 134
5.1 The Initial State . . . 136
5.2 Axiomatization . . . 136
5.3 Message Queue Model . . . 139
6 The Main Invariant . . . 140
7 Implementation . . . 141
7.1 Embedded Language for Algorithm . . . 141
7.2 The []-Operator . . . 141
7.3 Proof Tactics . . . 141
8 Proof . . . 142
8.1 Statistics . . . 142
8.2 Tactics . . . 143
8.3 Theorem Provers . . . 143
8.4 Proof Example . . . 145
8.5 Proof Procedure Summary . . . 146
9 Conclusions . . . 147

Paper 6 – Finding Counter Examples in Induction Proofs 149
1 Introduction . . . 151
2 Verification Method . . . 154
2.1 Failed Proof Attempts . . . 155
2.2 Identifying the Categories . . . 156
3 Finding Counter Examples by Random Testing . . . 157
3.1 QuickCheck . . . 157
3.2 Trace counter examples . . . 158
3.3 Induction step counter examples . . . 159
4 Results . . . 161
4.1 Trace Counter Examples . . . 161
4.2 Induction Step Counter Examples . . . 162
5 Discussion and Conclusion . . . 166

Appendix – Invariants, Axioms and Definitions for the Leader Election Algorithm Proof 169
A.1 Appendix Overview . . . 169
A.2 Predicates . . . 169
A.3 Functions/Constant Symbols . . . 169
A.3.1 State-arrays - indexed by host . . . 169
A.3.2 Natural numbers . . . 170
A.3.3 State-names . . . 170
A.3.4 Global sets . . . 170
A.3.5 Messages . . . 170
A.3.6 Set functions . . . 170
A.3.7 Array functions . . . 170
A.3.8 Host function . . . 170
A.4 Axioms . . . 171
A.4.1 Axioms – stoller . . . 171
A.4.2 Axioms – con-sys . . . 172
A.4.3 Axioms – cons-snoc . . . 172
A.4.4 Axioms – arith . . . 174
A.4.5 Axioms – sets . . . 175
A.4.6 Axioms – twice-msg . . . 175
A.5 Invariants . . . 175

Acknowledgements

First I would like to thank my brilliant supervisor Koen Claessen. Koen has a virtually endless supply of ideas and has been very supportive in every aspect of my PhD studies. Koen was also the person who got me into verification and logics in the first place, now some six years ago!

Many thanks to Thomas Arts, co-author of two of the articles in the thesis, and Lars-Åke Fredlund, also co-author of two articles in this thesis. Thomas and Lars-Åke have provided much valuable knowledge from the Erlang community, and it has been a great pleasure working with them.

I am very grateful for the valuable comments and corrections to the thesis provided by my opponent Jaco van de Pol. His suggestions has improved the thesis in numerous places.

I am also grateful for the feedback to this thesis provided by Mary Sheeran, who has also given me good support as examiner and member of my PhD committee.

Thanks also to John Hughes for his support in the work with Erlang QuickCheck and Graham Kemp for being a member of my PhD committee.

Many thanks to the previous Formal Methods group, and newly formed Fuctional Programming group for the inspiring discussions and interesting seminars.

Many thanks to my outstanding office-mate and friend Emil Axelsson. We have shared an office for 4.5 years and have had a good time together. I wish him the best of luck with his ongoing thesis writing. Also thanks to my previous office-mate Magnus Björk, and to my good friend and colleague Andreas Larsson who reminds me when it is time for lunch. I am also grateful to everyone in the department, this is an enjoyable place work.

Finally, thanks to my fiancé (and soon to be wife), Elin, for her continual support and love!

Introduction

Computers are everywhere nowadays; they are part of almost every aspect of modern society. This means that we are becoming increasingly dependent on computerized systems, be it in cars, telecommunication, energy distribution or our daily work environment. Along with computers comes computer software, the programs that dictate what the computer should do. The flexibility of software-controlled systems, the constant need for new and more advanced products and business competition drive the development with an enormous force.

Despite its widespread use, software is almost never error-free. For a wide variety of reasons it is extremely difficult to construct 'perfect' software systems. Lyu's opinion is *"software is a systematic representation and processing of human knowledge"* [Lyu95], and perfect knowledge about a problem is rarely achieved. Abbott's opinion is not very encouraging either *"programs are really not much more than the programmer's best guess about what the system should do"* [Abb90]. An important aspect is that the more important a system is, the more costly, be it in terms of economy or some other measure (ultimately even human lives), is a failure. A US Congressional report from 2002 estimate that software errors cost the US economy $60 billion annually [Ins02]. At the same time, Sommerville reports that a normal software project spend 40% to 50% of its budget on testing and system integration [Som06].

Much has been improved since the first bug [Hop81], but software errors are still part of far too many computer systems. It is clear that there is a huge demand for methods and techniques to produce better software. And although construction of reliable software is a problem without an obvious solution, time, money, research and hard work are constantly invested in improving the situation.

This dissertation is focused on the problem of *verifying algorithm implementations*. The problem is studied in a *distributed* and *fault-tolerant* setting. The introduction tries to give a general perspective of software development issues together with relevant related work. It gives a gentle introduction to the leader election problem, explains the terms distributed and fault-tolerant, introduces the Erlang programming language, and discusses the task of implementing a formally verified algorithm. The introduction finally highlights the main contributions of the thesis and gives an overview of the rest of the thesis.

1 Software Development

Software development is *"the translation of a user need or marketing goal into a software product"**. The software industry is one of the fastest-growing industries. The 500 largest software companies had a total revenue of $394 billion in 2006 [Des07] and almost 3 million employees. Software development is also a large research area, mainly because it is inherently difficult to construct reliably working software. There are several reasons for this:

- It is hard to specify what the *correct* behavior is.
- Systems are often very complex, and developers lack the tools and methods to deal with this complexity.
- The programming languages and development tools do not give the developer enough support.
- Some underlying (theoretical) problems are still awaiting a solution, while others are unsolvable.

Software development research is often focused around the problem of verifying that a given piece of software behaves correctly; however, before that problem can be attacked, we need to specify the meaning of 'correctly' in this setting. That is, we need some sort of *specification* to precisely express the intended behavior of the system [PST91]. In practice such a specification often also has a second purpose, it can be seen as an agreement between the contractor and the customer. Nevertheless, having settled for a specification, another question quickly arises: How do we know that the specification is correct? This is an equally important, and difficult, question that must not be overlooked [CAB+98].

Once a sufficiently detailed and trustworthy specification has been agreed upon, there are two main approaches to check that the system conforms to the specification: *informal software verification* and *formal software verification*. The two approaches are not orthogonal and there is a certain degree of overlap.

1.1 Informal Software Verification

With informal verification we usally mean methods that give an *incomplete* judgement about the *actual system*. The most common view on informal verification is *software testing*. Software testing is an operational way of checking software correctness (or rather software in-correctness, since one is trying to expose errors) by *executing* the software and *inspecting* its behavior [Bei90]. Tests are usually executed in a controlled environment to a system under test (SUT) [Tre92]. The standard way to conduct testing, is to create a *test suite*, where program input is specified together with its expected outcome. Using the test suite, the program is executed with input from the test suite and the results are compared with the expected results [Mye79, Bei90].

*http://en.wikipedia.org/wiki/Software_development

There are also more formal approaches to testing [TB99], where formal theories are used to increase the effectiveness of testing. One such approach is *model-based testing*, where tests are generated from a formal specification of the SUT [DJK+99, OA99, FTW06].

1.2 Formal Verification

In formal verification, logic-based methods are used to give a *complete* judgement about a *model* of the system. There are a variety of different approaches to formal verification, including *(automated) theorem proving* [RV01] and *model checking* [CGP00]. A formal verification technique proves that the system fulfills the system specification. In theorem proving the system is modeled in some suitable logic, and a proof of the specification is constructed using deductive reasoning [Hoa69]. In model checking a finite state representation of the system is used. The specification is often expressed using *temporal logic* [HR00], and the model checker test whether the temporal logic formula is true for the state space [CGP00]. The result of a formal verification attempt is either a proof of correctness or some sort of failed proof attempt. The second could be regarded as a failure, however, much information can be gained also from a failed verification. For example in model checking, the result of a failed verification is a *counter example*. The counter example is an explicit example of how the system does not fulfill the specification, and from this information it is often possible to deduce what is wrong with the system (or specification).

2 Distributed Systems

A non-theoretical definition of a distributed system (borrowed from Tel [Tel00]) is *"An interconnected collection of autonomous computers, processes or processors."* The computers, processes or processors are referred to as the *nodes* of the system. Note that processes may play the role of nodes in a system. Thus the definition does not exclude software running on a single hardware installation.

There are several reasons for using a distributed computer system. The most common reasons are:

- **Performance** – If a computation can be performed in parallel, it is possible to increase the performance (measured in time to result) substantially by using a distributed system.

- **Reliability** – In critical systems a distributed system is used for fault-tolerance. A simple example could be one machine (or processor) doing the work with a second machine (or processor) replicating the first and taking over in case of failure.

- **Physical distribution** – Physically distributed devices could make distributed solutions necessary.

- **Resource sharing** – The ability to share resources (computation power, storage, peripheral units, etc) could be a reason for building a distributed system.

It is clear from the list above that there is a lot to gain in some situations. However, there is also a price for the improved capacity, namely an increased complexity in the software running on the distributed systems. The possibility of running several operations in parallel might lead to *non-deterministic* programs; meaning that the result of the program depends on the order in which operations happen to finish. As a result of this added complexity, it is even harder to verify distributed software [HM85]. This effect is particularly evident when one tries to reproduce errors, since an error might only be detectable in a particular order of events [CT91].

3 Fault-Tolerance

A fault-tolerant system is designed to be able to continue its operation when some part of the system fails. If the system continues only at a reduced level, it is said to be *gracefully degrading*. Fault-tolerance is particularly desirable in high-availability or life-critical systems. Three important questions to ask regarding the use fault-tolerance are:

1. How important is the system?
2. How likely is the system to fail?
3. What is the cost of fault-tolerance?

Fault-tolerance for software systems comes in many different flavors. Fault-tolerance can be achieved by anticipating exceptional conditions and designing the system to handle such situations, or aim for self-stabilization so that the system always converges towards an error-free state. Another way to achieve fault-tolerance is to introduce *redundancy* (having several instances of the system, switching to one of the remaining instances in case of a failure – *fail-over*) or *replication* (having several instances of the system all running in parallel, choosing the correct result by voting). For further discussion of fault-tolerance, we refer to a report by Torres-Pomales [TP00].

Naturally, additions to achieve fault-tolerance increase the complexity (and thus in the end the cost) of the system. The increased complexity also means that verification of fault-tolerant systems is often harder than verification of systems without fault-tolerance. There are platforms, such as Erlang OTP [Tor97], FT Concurrent C [CGR88], and to some extent Ada [Bar95], that are designed with fault-tolerance in mind. Using such a platform makes the complicated task of achieving fault-tolerance somewhat easier.

4 Leader Election

The problem of leader election was first posed by LeLann [LeL77] and can be formulated generally as:

> Given a set of participating processes, the processes should among themselves designate a single process as the leader and all other processes should recognize the leader.

The leader election problem has been studied (and solved) for a variety of different assumptions. Usually the basic assumptions are that each participant has a *unique identifier* and that the goal is to choose the member with the largest (or smallest) identifier as the leader. There are, however, different *fault assumptions*:

- *Processes* may or may not fail
- *Communication links* may or may not fail
- *Messages* may or may not be corrupted

In addition to the different fault assumptions there are also different *time assumptions*. Models are usually divided in *synchronous* [BB02] and *asynchronous* [CF99] time models. Depending on the different assumptions the leader election problem is more or less complicated to solve. Below we describe two different assumption settings as well as proposed solutions.

4.1 Ring Networks

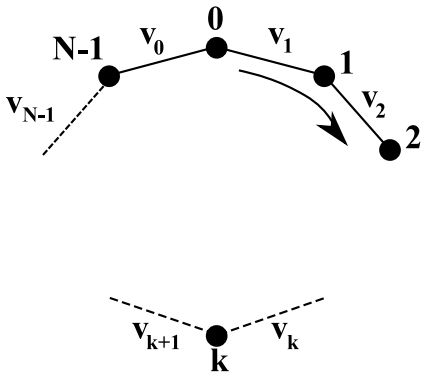


Figure 1: Unidirectional ring network

A ring network with N nodes is the graph with nodes 0 to $N-1$ where node i has edges v_i and v_{i+1} (all indices are calculated modulo N), see Fig. 1. Ring networks are frequent in the study of distributed algorithms, mainly because of their simplicity. However, there are also some physical networks, for example the well known *Token Rings* [Tan96], arrange their nodes in a (virtual) ring structure.

For convenience there is assumed to be an underlying layer responsible for reprogramming the structure in case of node failure, thus maintaining the ring structure. One important aspect is the difference that often exist between the theoretical settings explored in algorithm papers and the "real" settings where the algorithms are used. (See also Sect. 6.1, in which the *semantic gap* between theoretical and practical settings is further discussed.)

The leader election problem was originally posed for the *unidirectional* ring network (in which messages can only pass in one direction through the edges) context by LeLann [LeL77]. LeLann also gave a solution with a message complexity in terms of number of messages of $\mathcal{O}(N^2)$. LeLann's solution was slightly improved by Chang and Roberts [CR79]. For *bidirectional* ring networks (messages are passed in both directions) Franklin described an $\mathcal{O}(N \log N)$ solution [Fra82]. Peterson [Pet82] and Dolev, Klawe and Rodeh [DKR82] independently adapted Franklin's algorithm to an $\mathcal{O}(N \log N)$ solution also for unidirectional ring networks. It has been proved that $\mathcal{O}(N \log N)$ is the lower bound for unidirectional ring networks. Naturally the bounds are only applicable in a stable situation, when nodes do not cease to work during the election.

The algorithms mentioned above can cope with different failures, for example LeLann discusses how to cope with crash failure (nodes stop working, but do not start again). There are also even more robust algorithms, namely the family of *self-stabilizing* algorithms. A self-stabilizing algorithm is able to fulfill its task regardless of which state it happens to start in. (Imagine a ring network where the nodes are hit by ionizing radiation.) Note that this is a substantially stronger requirement than tolerating node failures and communication failures, and naturally an algorithm satisfying these conditions is more complex. A first self-stabilizing protocol for ring networks was presented by Burns and Pachl [BP89] and an improved version was suggested by Huang [Hua93]. Another variation is *anonymous* ring networks, where processes do not have unique identities. For anonymous ring networks, no terminating deterministic algorithm exist [Ang80]; however, Itai and Rodeh has shown that probabilistic methods can break symmetry in anonymous ring networks [IR90].

4.2 The Bully Algorithm

In this scenario we consider an arbitrary, fully connected network instead of the simpler ring network. We also define beforehand the set of participating nodes. Using this model the selection task seems trivial; simply select the node with the smallest (according to some ordering) identifier. However, since nodes are allowed to fail and restart at any point in time, the problem is far from being trivial.

The *Bully Algorithm* was presented by Garcia-Molina [GM82] as a solution to this problem. The algorithm got its name because in the election process nodes with high priority force nodes with lower priority into accepting them as the leader. (Note, Garcia-Molina used the term *coordinator* instead of leader.) The Bully algorithm is used, with some modifications, in our leader election implementation.

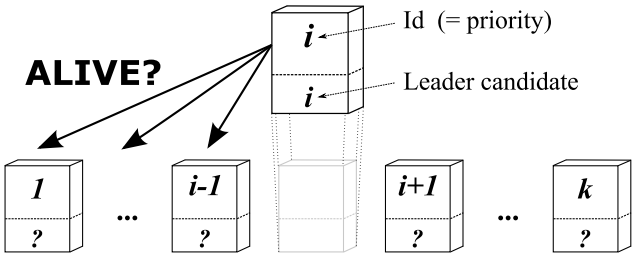
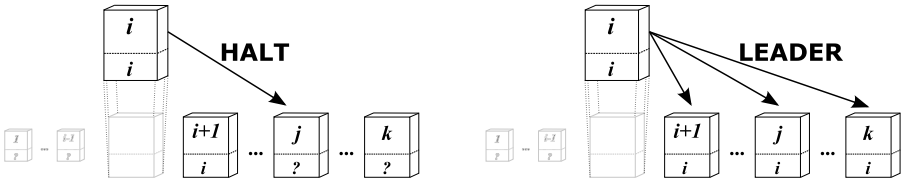


Figure 2: Election phase 1



(a) Part 1 (b) Part 2

Figure 3: Election phase 2

Assume that the node with identification number i starts an election. (There are several reasons for starting a new election; the old leader died, the process just recovered or the last election was aborted since the candidate leader died.) The election protocol is divided into two phases. In the first phase, the node communicates with nodes that have a higher priority and in the second it communicates with nodes that have a lower priority. First, as shown in Fig. 2, node i tries to contact all nodes with higher priority. If any of the nodes with higher priority is alive, node i gives up its bid to become leader and waits for communication from that node. (If i does not hear from this node in a while, it should again restart the election process). If no node with higher priority is alive, node i continues with the second phase of the algorithm. In the second phase, node i contacts all nodes with lower priority (in priority order) to inform them that node i intends to become the leader. This is done in a two-step process by first force all nodes with lower priority into a state where they are ready to accept the new leader, shown in Fig. 3(a). Thereafter, to actually become the leader, node i sends another message to all nodes with a lower priority, illustrated in Fig. 3(b). (To get a better intuition of why this works one can read the informal introduction of the algorithm in Garcia-Molina's paper [GM82].) If i itself has the highest (or lowest) priority, the first (or the second) phase is trivial.

4.3 Verification of Leader Election Algorithms

The leader election problem has been covered in many different papers. Numerous solutions to the problem exist, differing in assumptions about network topology, message passing, identification of participants, etc. [BKKM96, DIM97, Sto00,

ADGF01]. Published leader election algorithms often include a correctness proof [Sin96, Sto97]. For other algorithms verification has been performed separately; ring network algorithms of LeLann's [LeL77] and Chang and Roberts [CR79] have been specified and verified by Garaval and Mounier [GM96] using the LOTOS specification language and the CADP toolbox. Fredlund et al. [FGK97] performed, using process algebra, a similar specification and verification effort for the ring network algorithms by Peterson [Pet82] and Dolev et al. [DKR82]. Devilliers et al. used an I/O automata model to verify the leader election part of the IEEE 1394 high performance serial multimedia bus protocol (FireWire) [DGRV00, Rom01]. Usenko studied the HAVi leader election protocol using both SPIN and μCRL [Use99].

One problem with implementations of algorithms is that, as discussed in [ACHS05], implementations rarely follow the algorithm exactly. Therefore, the formal verification of the algorithm does not easily transfer to the actual implementation.

5 Erlang

Erlang [AWWV96, Arm07] is a dynamically typed functional programming language. Erlang was developed by Ericsson, a Swedish telecommunication company, and consists of a small functional core together with powerful constructions for concurrency. Erlang is especially suited for implementing fault-tolerant distributed systems. The language has built-in primitives for operations such as process creation, message passing and process supervision. The development of Erlang was driven by demands for fault-tolerance and also by demands to have continuously running systems that cannot really be taken down to be able to perform software upgrades and add functionality. Therefore, Erlang also includes functionality for hot code replacement (that is replacing code at runtime without stopping the system).

Most software written in Erlang is running in a distributed environment. In Erlang terminology: a distributed system consists of *nodes*, which communicate over a network. Each node contains multiple light-weight *processes* that are separated in memory. Processes use *asynchronous message passing* as their only method of communication. The message passing is implemented with a per-process mailbox, which is always ready to receive a message. An expressive pattern matching syntax gives a process control over message retrieval from its mailbox [Arm03]. Further, Erlang supports process *linking*, which means that a process A can obtain a *link* to a process B. If process B fails, the linked process A gets a notification message about B's failure [Wik94].

OTP (Open Telecom Platform) is a development platform for building telecommunications applications, and a control system platform for running them [Tor97]. OTP is based on Erlang, and was originally targeted only for telecommunications. However, the OTP system architecture includes many useful tools (ASN.1 compiler, SNMP support, Trace tool, Debugger, Profiler, etc.) and building blocks (Mnesia real-time DBMS, OS monitoring, Erlang Virtual Machine, Web server, etc.), which have proved to be very useful also in non telecommu-

nication applications [Erl07]. OTP also specifies certain *generic behaviors*, such as client-server communication, finite-state machines and process supervision hierarchies. These behaviors are widely used, and many applications are built upon them. Because of their wide usage and structured implementation, several program verification initiatives have been targeted against the generic behaviors [AF02, ABS04].

5.1 Erlang Today

Erlang was released as *open source* by Ericsson in 1998, and since then its popularity has slowly increased. The concurrency oriented nature [Arm03] and the (mostly) transparent distribution has made Erlang a good candidate for writing efficient distributed systems as well as for experimenting with distributed systems in research. Lately the introduction of multi-core systems has further strengthened the position of Erlang. Since release 5.5/OTP R11B, Erlang has built-in support for *Symmetric Multi Processing* (SMP) and the SMP support is completely transparent. How much is gained, i.e. how much it can take advantage of the multi-core technology, is of course highly dependent on the particular application. However, tests shows that existing Erlang applications, without any modifications, get significantly improved performance [Eri06].

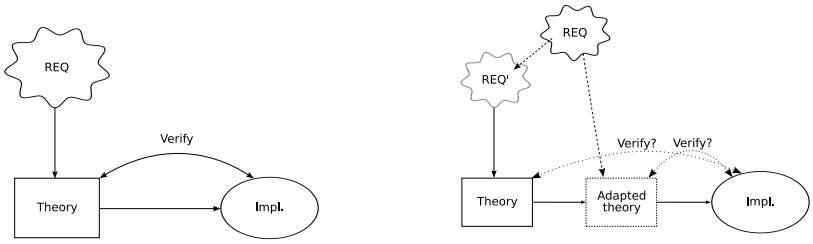
5.2 Verification of Erlang Programs

There are many different approaches for testing Erlang programs. One interesting testing technique is automatic test case generation [BJ03], that has been used for industrial applications written in terms of OTP components. Another approach is to use the built-in tracing system, and perform analysis of the collected traces [AF02]. Paper 1 in this thesis is an extension of this approach. It is also possible to use flow graphs to measure test coverage [Wid04], but that work focuses on the sequential part of Erlang and is not applicable to distributed systems. Random testing [Ham94], especially in the QuickCheck implementation [CH02, AH03], is a powerful tool for simple and effective testing of (possibly distributed) software. When using QuickCheck, properties are specified for the program. QuickCheck then tests that the properties hold in a large number of randomly generated cases.

There are also some approaches to formal verification of Erlang programs. One formal verification technique is to translate Erlang into μCRL [ABS04] and then model check the μCRL-model. This approach has been applied successfully to a resource locker [ABD04, EFD05]. A similar approach using μ-calculus is [DFG98], which introduces both a logic and a proof system for specifying and proving properties of distributed systems. The "*Verification of ERLANG Programs*" project [DF98, FGN01, FGN$^+$03] resulted in the theorem prover, Erlang Verification Tool (EVT). EVT assists in making proofs about Erlang programs. Huch uses a rather different approach [Huc99] (later refined in [Huc01] and [Huc02]), where abstract interpretations are used to reduce the size of the state space before applying model checking. Noll has tried both a rewriting logic implementation of Erlang [Nol01], and modeling Erlang in π-calculus [NR05].

6 Implementing a Formally Verified Algorithm

In practical software development one is often faced with an algorithmic problem, and for most such problems there are books and papers describing potential solutions. To use a (formally) verified algorithm seems to be a good strategy; however, algorithm implementation is normally a rather complex process.



(a) Ideal situation (b) Normal situation

Figure 4: Implementing algorithm from theory

In the *ideal situation*, see Fig. 4(a), we have the requirements (REQ) and find an algorithm/theory that fulfills the requirements. Having made an implementation we must then ensure that the implementation is correct with respect to the theory. The key observation is that we can benefit from the fact that the theory is verified. Still, this is in itself not an easy task. However, in the *normal situation* we are even worse off, see Fig. 4(b). Firstly, we often have to (slightly) adapt the requirements in order to find a suitable algorithm (*semantic incompatibility*). Secondly, the original requirements might make adaptations of the algorithm necessary (*context adaptation*). Thus, in the resulting situation it is unclear what to verify. Nevertheless, we argue that we could still benefit from the verification of the original algorithm, although doing so requires some extra care. To summarize, the two main issues are:

1. **Semantic incompatibility**, algorithms assume a semantic model of the execution environment that is not compatible with (nor can be adapted to) the actual execution environment.

2. **Context adaptation**, requirements force the addition of features or even slight modifications of the algorithm during implementation.

6.1 Semantic Incompatibility

Algorithms are presented for an often implicit programming model. There are incompatibilities for sequential algorithms. For example the original Quicksort algorithm [Hoa62] assumes an imperative implementation language, which is rather different from an object-oriented language or a functional language. For concurrent and distributed algorithms this is even more evident. For example, if the

underlying assumption is a shared memory architecture, an implementation in a message passing environment is quite likely to introduce errors that do not stem from the algorithm. Similarly, an algorithm for a ring network could be implemented for a TCP/IP based network (which is often similar to a fully connected network). One could even simulate a ring on top of such network; however, some reasoning is required to ensure that the simulated ring indeed has the properties assumed in the description of the algorithm. Strictly speaking, the correctness of such transferred implementations is an open issue and referring to a proof of the underlying idea creates false trust.

Of course, it is impossible to overcome all semantic incompatibilities. New programming languages, components, operating systems, network technology, and other technical innovations cannot be foreseen. Moreover, one cannot possibly describe one algorithm for all the existing semantic models.

6.2 Context Adaptation

Software requirements are usually not expressed in terms of 'implement algorithm X.' Algorithms may form part of the total solution. Given a set of requirements, it is often hard to recognize the possibility to use a certain existing algorithm. A second challenge is to find a version of that algorithm that fits the requirements best.

Implementing algorithms practically always requires some modifications that seem not to influence the algorithm, but which can still be a source of errors. For example, one could be required to implement an algorithm using the file system as storage medium instead of primary memory. Immediately, one is forced to test whether files are already open, whether one has write access and so on. These small, rather straightforward changes may easily introduce an error, although the algorithm is correct.

6.3 Verifying the Implementation

These issues are certainly not new. The gap between formal models and implementations has been known for a long time. In the seventies, Knuth made a famous remark about a dequeing algorithm *"Beware of bugs in the above code; I have only proved it correct, not tried it"* [Knu77]. That is, given that we have a proof of correctness, which might be rare in general, there is still a gap and there are several possible ways to bridge the gap.

- *Informal reasoning* – One can do as Knuth actually did. He reasoned about the algorithm, and mimicked a formal proof in the lines of code written (while having the semantics of the pseudo language in mind).

- *Testing* – One can do as Knuth (more or less) suggests, write a few test cases to check that no obvious mistakes are made.

- *Program analysis* – One can analyze the source code, by using slicing, abstraction, etc. [Wei81, CGL94]

- *Model checking* – One can implement the semantics of the programming language in a model checker [HS02, FS07].
- *Theorem proving* – One can implement the semantics of the programming language in a theorem prover [BM75, HJ00, FGN+03, ABB+05].
- *Code generation* – One can automatically generate the implementation from a more high-level description [BFVY96, Har01].
- *Property based trace testing* – One can generate a part of the state space and prove properties about the generated part. Errors found in this way are real errors, which is good. Provided that the test vectors are realistic (which can be hard to assure) most errors can be caught with this method. [AF02, HBUP03, ACS05]

Informal Reasoning

During the implementation of an algorithm, there is some implicit informal reasoning; the developer reasons like: 'Yes, I am implementing what is proved/presented in this paper'. Very often this type of reasoning is not regarded as part of the verification process. Nevertheless, it is very important to realize that this reasoning happens, and that it is likely to influence the rest of the verification. Take the testing process as an example, the developer (if involved in the testing) is very likely to focus on parts of the system that were hard to implement.

Testing

Everyone who has done software development has probably some intuitive picture of what constitutes a good software testing approach. Most developers would also agree that it is in general a very difficult task to carry out a good testing effort. Another well known fact is that testing concurrent/distributed systems is more difficult than sequential, single-process, systems [MH89].

When discussing implementations of (formally) verified algorithms (with possible adaptations), we focus on what is gained in the test process by having the algorithm description. In a standard testing environment (with an ordinary testbench and hand-made or generated test cases) we do not gain very much. Firstly, the algorithm description is not likely to describe how to construct test cases or stimulate the system. Secondly, in the case of a distributed system, the properties stated in the algorithm verification are often *global properties* that cannot be easily checked for a test run of the system. In a formal testing environment the situation is a bit better. For example, algorithm properties could help in constructing a model for model-based testing [DJK+99].

Model Checking

If the algorithm we have implemented is model checked (or proved correct with a theorem prover) it is tempting to also try to formally verify the implementation. However, in general this is not feasible. It is both expensive (in terms of time)

and complex (in terms of education/level of expertise). The main complications are:

- **Programming language semantics** – The model must not only model the actual implementation, but also the semantics of the implementation language. (Simplifications are often possible, but not without the risk of errors, see *abstractions* below.) It is preferable to push the model as close to the actual implementation as possible; however, that means more details and in the end a larger state space to check. There are several successful projects, where models are automatically extracted from program code, for example MODEX (Ansi-C to Promela model) [Mod] and Bandera (Java to Promela model) [Ban].

- **State space explosion** – The hardness of the model checking problem is often governed by the type of system at hand and the level of detail. Ordinary sequential programs can often be coped with, but distribution and/or fault-tolerance often causes enormously large state spaces [Val98]. Many different techniques, such as abstractions, partial order reductions, symbolic representations and distributed approaches, have been developed to handle the state space explosion problem [McM93, CGL94, Pel98, BBS01, BLvdPW08]. Nevertheless, it is still a problematic issue.

- **Abstractions** – To handle a sufficiently detailed model (maybe including distribution and fault-tolerance) it is often necessary to use abstractions. An abstraction is essentially a mapping from concrete states into abstract states, where (in order to reduce the number of states) several concrete states are mapped to the same abstract state [CGL94]. Designing good abstractions is far from trivial, and incorrect use of abstractions may invalidate the model checking results. Abstractions are further discussed in the next section 7.

Theorem Proving

Program verification by theorem proving is a high-end, very complex, approach. While theorem proving is quite common in algorithm design, it is far less used in large (industrial) software projects. (Despite the complexity, there are a number of success stories, such as verification of a floating point arithmetic unit [Har99] and verification of a resource locker[ABD04].) The main reasons are:

- **Non-reusable proofs** – A formal proof often contains some sort of proof object or proof script. Although it is easy to re-run a proof script if the conditions change, it is not very likely that the same reasoning goes through unchanged. Because of the high level of detail, even a small modification can require substantial changes in the proof. The same things can be said for informal reasoning; however, the risk for errors after changes in informal reasoning is bigger. The lack of formality means that it is easy to overlook the consequences of a minor change.

- **Lack of automation** – Theorem proving is hardly automatic today. The most successful approaches [Har99] are using *(semi-)interactive* methods and theorem provers such as HOL, Isabelle, PVS, Key and ACL2 [NPW02, ORS92, ABB+05, KM08, BKM96]. The interaction require a high level of education for the person performing the verification.

- **Programming language semantics** – Since we are proving correctness of the implementation one must also take into account the (possibly complex) programming language semantics. Several theorem proving approaches consist of customized proof systems that incorporate the programming language semantics [BM75, ABB+05, Moo03, FGN+03].

Property Based Trace Testing

Property based testing is a specialization of testing, where the tests are driven by software properties [FB97, SC96]. Property based testing roughly consists of two phases, (1) specifying properties and (2) testing that the system fulfills the properties. One commonly used property based testing framework is QuickCheck [CH02, Hug07], which is a property based tool for random testing.

Property based testing is more formal than ordinary testing. At the same time it is still far more lightweight than model checking and theorem proving since the only formal part are the properties. We think that property based testing goes hand-in-hand with formally proved algorithms. That is, property based testing should be especially suitable for testing implementations of formally verified algorithms, since it is hopefully rather easy to extract properties from the algorithm description [ACHS05].

7 Abstractions

Systems gradually are getting larger and more complex, at the same time the verification task is simultaneously getting harder. Since model checking generates (either explicitly or symbolically) all possible system states, it is crucial not to have a too many system states. (Hardware and algorithm development has pushed the state space size limits quite some way, but still, the size of the state space is a main concern.) To cope with a detailed model and perhaps also distribution and fault-tolerance it is almost always necessary to make abstractions. In Fig. 5 we see an example of a simple abstraction. On the left side we have the concrete state space, and the corresponding abstract state space is shown to the right. The abstraction maps concrete states (here S1,...,S4) into abstract states (here AS1 and AS2, where S1 and S4 maps to AS1 and S2 and S3 maps to AS2). An important observation is the fact that every *path* (that is a sequence of states obtained by a series of transitions) in the concrete state space is represented by an abstract path in the abstract state space. A corollary of this is that a safety property that holds for the abstract state space also holds for the concrete

state space† [CGL94]. The converse is not true, if a property is violated in the abstract state space, the counter example may possibly not have a corresponding counter example in the concrete state space. This is known as a *false negative*, a situation in which a positive result is falsely reported as a negative result. False negatives are the result of the information loss that occurs when reducing a large concrete system into a smaller abstract system. The situation is also known as *overapproximation*. To avoid such false negatives, one needs to refine the abstraction. The goal is to not introduce the spurious behavior in the abstract state space. In some situations abstraction refinement can be automated given a false counter example [CGJ+00].

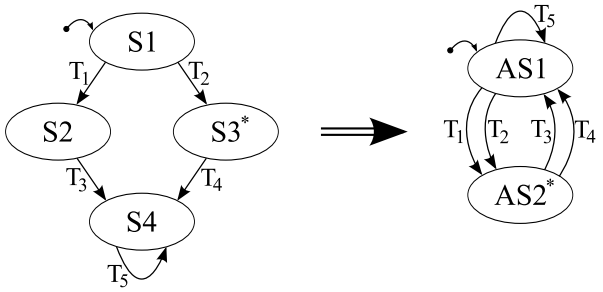


Figure 5: Abstraction example

When it comes to handling abstractions, there is a difference between symbolic state model checkers and explicit state model checkers. Using a symbolic state model checker it is often possible to handle the abstraction algorithmically, by translating the abstraction to a symbolic operation [McM93]. This can, however, be a complex and very computationally expensive operation. In an explicit state model checker, such as McErlang [FS07], it is generally impossible for larger problems to compute an exact abstraction like in Fig. 5. In practice that would mean that all states have to be constructed, which we wanted to avoid by using the abstraction in the first place. One solution is instead to use an *approximation* of the abstraction, where abstract states are computed on-the-fly during model checking. Whenever the model checker explores a transition, the abstract representation of the resulting state is computed and compared to already visited states. In principle this means that for each set of concrete states mapping to the same abstract state, the first encountered state in the set is selected as a representative for the whole set. The abstract state space in Fig. 5 has two such sets, {S1,S4} and {S2,S3}. There are problems with such an approximation; the problems stem from the fact that this particular abstraction approximation is an *underapproximation*. The result is that we can get, as shown in the example, *false positives*. A false positive is a situation where a negative result is falsely reported as a positive result. In an explicit state model checker this means that a bad state might go undetected. The following example illustrates this problem.

†This is more general, the reasoning holds for all LTL properties [HR00] since they implicitly quantify over *all paths*, however, it does not hold for example for all CTL properties [HR00].

Example: In the state spaces in Fig. 5 bad states are marked with an asterisk (*). We see that S3 is a bad state, and thus also AS2. In the initial state S1, the model checker picks either T_1 or T_2 first, and depending on the choice either S2 or S3 is going to represent {S2,S3} in AS2. I.e. if T_1 is explored first, and thus S2 is the representative of {S2,S3}, then later when T_2 is explored (leading to S3) S3 is regarded as already visited and thus the bad state is missed.

8 Contributions

In this section we summarize the main contributions of the thesis. For each contribution there is also a pointer to the paper(s) describing the contribution in greater detail:

- *Finding errors in an existing Erlang implementation* – We found, analyzed and explained two bugs in an existing Erlang implementation. The bugs were non-trivial in that they only occurred in particular configurations. *[Paper 1]*

- *Trace abstractions* – We developed methods for analyzing trace data, we presented a language for expressing abstractions as well as means to check LTL-properties [HR00] for a given (abstract) trace. *[Paper 1]*

- *Erlang implementation of new leader election algorithm* – Having failed to repair the existing leader election implementation, we implemented a new algorithm. To the best of our knowledge, the algorithm is a novel leader election algorithm. The algorithm is based on Stoller's algorithm with a few Erlang-inspired modifications. The requirements on the implementation were slightly different from the original algorithm, where the main change was to sacrifice message complexity for fewer re-elections. In the standard Erlang setting, sending messages is cheap, but changing the leader is not. The implementation has withstood very thorough testing and a variety of model checking approaches. The implementation is available as open source. *[Paper 1, Paper 2]*

- *Distributed Erlang semantics* – There existed an Erlang semantics for the single node case, but as the bugs in the leader election implementation clearly showed there are significant semantic differences between the single node and the distributed case. We present a small step operational semantics for distributed Erlang in the paper [SF07]. (The paper [SF07] is a revised version, where we have made important additions and corrected a few minor errors, of an earlier paper [CS05].) *[Paper 3]*

- *Model checker* – We have developed our own customized mode checker *McErlang*. Although the model checking was not successful for the leader election example, it has worked out well for other applications. The Erlang model checker McErlang has been used in several case studies with good results. *[Paper 4]*

- *Proof of Stoller's algorithm* – Even state of the art model checkers could not handle the enormous state space produced by the leader election algorithm. Instead, we used a FOL model and have proved the important safety property "There is never more than one leader" for Stoller's leader election algorithm using automated theorem provers with some interactive steps. It is, however, still work in progress to prove the correctness of the adapted version of the algorithm present in the new implementation. *[Paper 5]*

- *Algorithm verification method* – The verification method used to prove Stoller's leader election algorithm is more general, and should be applicable to other similar algorithms. Since we have only tested with two different examples (Stoller's leader election algorithm and a toy example with resource allocation), we can only speculate on the true generality, but there are no explicit limitations. *[Paper 5]*

- *Counter examples in inductive proofs* – One specific part of the proof methodology is the search for counter examples. The situation that is handled is when one is trying to use a theorem prover to inductively prove that an invariant holds. The problem is the case when the invariant cannot be proved (i.e., the theorem prover does not give an answer). Then it is unclear how to proceed since the situation could be either one of four possible cases. The contribution is two-fold: we have identified and explained the four different cases, and we have implemented and evaluated a QuickCheck driven search for counter examples in two of the four cases. (One case remains future work. In the fourth case, where the theorem prover is unable to prove a provable formula, it is rather unnecessary to search for a counter example.) *[Paper 6]*

9 Thesis overview

The thesis is based on six papers, each discussing a topic in software testing/model checking/formal verification. The thesis is divided into seven main sections; one section for the introduction and one section each for the six papers.

Paper 1 – Semi-Formal Development of a Fault-Tolerant Leader Election Protocol in Erlang The background was a promising testing technique for Erlang [AF02]. The main idea is to run the tested implementation with generated stimuli; thereby creating *traces* of events and states. Then, using an *abstraction function* specified by the user, we generate abstract state transition diagrams of the system. In the paper we improve the representation of abstraction functions by introducing a separate language for expressing abstraction functions. We also improve trace collection and add LTL-property checking for the abstract traces. The improvements are tested on two different implementations (one previously existing and one new) of a leader election algorithm. The described method is generally applicable to Erlang programs; in particular to software written using OTP Erlang component library [Tor97].

Paper 2 – A New Leader Election Implementation In paper 1 we demonstrate that the existing leader election algorithm [Wig03] is faulty. We were unable to repair the this implementation in a satisfying and correct way. Therefore we decided to write a new implementation. The paper describes the algorithm used in the implementation, which is an adapted version of Stoller's leader election algorithm [Sto97]. The paper explains the adaptations, and the reasons for the adaptations, in detail. The paper also describes the extensive testing applied to the new implementation.

Paper 3 – A More Accurate Semantics for Distributed Erlang In the conclusion of paper 1, we argue that the errors found in the existing leader election implementation only occurs in a distributed setting. This implies that there is a fundamental difference between the single-node and the multi-node setting. However, this difference could not be modeled by the existing formal semantics for Erlang. We used Fredlund's Erlang semantics [Fre01], which accurately model single-node systems, as a starting point and extended it to a semantics for distributed Erlang. The claim is that the distributed semantics is intuitive and correctly models the behavior of a distributed Erlang system. The paper explain the semantics in detail.

Paper 4 – McErlang: A Model Checker for a Distributed Functional Programming Language Earlier attempts to model check Erlang systems, for example [ABS04], were based on Fredlund's single-node semantics [Fre01]. The paper introduces McErlang, a model checker based on the distributed Erlang semantics. McErlang has full Erlang data type support, support for general (multi-node) process communication, node semantics, fault detection and fault tolerance. McErlang can verify programs written using the OTP Erlang component library. McErlang is itself implemented in Erlang and also uses Erlang as its specification language. This 'all in Erlang'-approach is shown to be beneficial, especially important is the possibility to treat executable models interchangeably as programs and data.

Paper 5 – A Semi-Automatic Correctness Proof Procedure applied to Stoller's Leader Election Algorithm Although McErlang was a success, it could not handle the leader election implementation. (Except for, for unconvincing bounds on the number participants, etc.) We also tried a few other model checking approaches, including model checking using SPIN [Hol97], with negative result. Instead we used a first-order logic model and automated theorem provers to prove safety properties about the Stoller's leader election algorithm. The paper describes both the proof procedure and all the details of the actual proof.

Paper 6 – Finding Counter Examples in Induction Proofs This paper is closely related to the proof procedure in paper 5. The most difficult part of using the proof procedure is to design the needed invariants. In particular problem arises when an invariant cannot be proved. The situation could be either one of four cases: (1) the invariant is invalid, (2) the invariant is valid, but too

weak, (3) the invariant is valid, but the current axiomatization of the background theories is too weak, and (4) the invariant is valid and should be provable, but the theorem prover does not have enough resources to do so. Paper 6 describes the implications of each case, and introduces a counter example search to reveal cases (1) and (2). This search for counter example is similar in spirit to the introduction of random testing in Isabelle [BN04], and we claim that it increases the productivity significantly.

References

[Abb90] R.J. Abbott. Resourceful systems for fault tolerance, reliability, and safety. *ACM Comp. Surv.*, 22(1):35–68, 1990.

[ABB+05] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hhnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P.H. Schmitt. The KeY tool. *Softw. and Systems Modeling*, 4(1):32–54, 2005.

[ABD04] T. Arts, C. Benac Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *Int. J. Softw. Tools Technol. Transf.*, 5(2):205–220, 2004.

[ABS04] T. Arts, C. Benac Earle, and J.J. Sánchez Penas. Translating Erlang to mCRL. In *Fourth Int. Conf. on Application of Concurrency to System Design*, p. 135–144, Hamilton (Ontario), Canada, June 2004. IEEE Computer Society.

[ACHS05] T. Arts, K. Claessen, J. Hughes, and H. Svensson. Testing implementations of formally verified algorithms. In *SERPS05: Proc. of the Fifth Conf. on Softw. Eng. Reserch and Practice in Sweden*, p. 103–112. Mälardalen University Press, 2005.

[ACS05] Thomas A. K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. In *LNCS*, vol. 3395, p. 140–154, Feb 2005.

[ADGF01] M.K. Aguilera, C. Delporte-Gallet, and H. Fauconnier. Stable leader election. In *Distributed Comp. 15th Int. Conf. DISC2001*, vol. 2180 of *LNCS*. Springer-Verlag, October 2001.

[AF02] T. Arts and L.-Å. Fredlund. Trace analysis of Erlang programs. In *Proc. of the 2002 ACM SIGPLAN workshop on Erlang*, p. 16–23. ACM Press, 2002.

[AH03] T. Arts and J. Hughes. Erlang/QuickCheck. In *Ninth Int. Erlang/OTP User Conf.*, Nov. 2003.

[Ang80] D. Angluin. Local and global properties in networks of processors (extended abstract). In *STOC '80: Proc. of the 12th annual ACM symp. on Theory of Comp.*, p. 82–93, New York, NY, USA, 1980. ACM.

[Arm03] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, Dec. 2003.

[Arm07] J. Armstrong. *Programming Erlang – Software for a Concurrent World*. The Pragmatic Programmers, http://books.pragprog.com/titles/jaerlang, 2007.

[AWWV96] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1996.

[Ban] Bandera – a model checker tool set for Java. http://bandera.projects.cis.ksu.edu/index.shtml.

[Bar95] J. Barnes. *Programming in Ada95*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[BB02] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Readings in hardware/software co-design*, p. 147–159, 2002.

[BBS01] J. Barnat, L. Brim, and J. Stříbrná. Distributed LTL model-checking in SPIN. In *SPIN '01: Proc. of the 8th Int. SPIN Workshop on Model Checking of Softw.*, p. 200–216, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[Bei90] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[BFVY96] F.J. Budinsky, M.A. Finnie, J.M. Vlissides, and P.S. Yu. Automatic code generation from design patterns. *IBM Syst. J.*, 35(2):151–171, 1996.

[BJ03] J. Blom and B. Jonsson. Automated test generation for industrial Erlang applications. In *ERLANG '03: Proc. of the 2003 ACM SIGPLAN workshop on Erlang*, p. 8–14, New York, NY, USA, 2003. ACM Press.

[BKKM96] J. Brunekreef, J.-P. Katoen, R. Koymans, and S. Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Comp.*, 9(4):157–171, 1996.

[BKM96] B. Brock, M. Kaufmann, and J.S. Moore. ACL2 theorems about commercial microprocessors. In *FMCAD '96: Proc. of the First Int. Conf. on Formal Methods in Comp.-Aided Design*, p. 275–293, London, UK, 1996. Springer-Verlag.

[BLvdPW08] S. Blom, B. Lisser, J. van de Pol, and M. Weber. A database approach to distributed state space generation. *Electron. Notes Theor. Comp. Sci.*, 198(1):17–32, 2008.

[BM75] R.S. Boyer and J.S. Moore. Proving theorems about LISP functions. *J. of the ACM*, 22(1):129–144, 1975.

[BN04] S. Berghofer and T. Nipkow. Random testing in isabelle/hol. In *SEFM '04: Proc. of the Softw. Eng. and Formal Methods, Second Int. Conf.*, p. 230–239, Washington, DC, USA, 2004. IEEE Computer Society.

[BP89] J.E. Burns and J.K. Pachl. Uniform self-stabilizing rings. *ACM Trans. Program. Lang. Syst.*, 11(2):330–344, 1989.

[CAB+98] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. *IEEE Trans. Softw. Eng.*, 24(7):498–520, 1998.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel Distrib. Syst.*, 10(6):642–657, 1999.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV '00: Proc. of the 12th Int. Conf. on Comp. Aided Verification*, p. 154–169, London, UK, 2000. Springer-Verlag.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[CGP00] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 2000.

[CGR88] R.F. Cmelik, N.H. Gehani, and W.D. Roome. Fault tolerant concurrent C: a tool for writing fault tolerant distributed programs. *Fault-Tolerant Comp. 1988. FTCS-18, Digest of Papers., 18th Int. Symp. on*, p. 56–61, 27-30 Jun 1988.

[CH02] K. Claessen and J. Hughes. Testing monadic code with QuickCheck. In *Haskell '02: Proc. of the ACM SIGPLAN workshop on Haskell*, p. 65–77, New York, NY, USA, 2002. ACM Press.

[CR79] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22(5):281–283, 1979.

[CS05] K. Claessen and H. Svensson. A semantics for distributed Erlang. In *ERLANG '05: Proc. of the 2005 ACM SIGPLAN workshop on Erlang*, p. 78–87, New York, NY, USA, 2005. ACM Press.

[CT91] R. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Softw.*, 8(2):66–74, 1991.

[Des07] J.P. Desmond. The software 500: Applications go worldwide. World Wide Web electronic publication, http://www.softwaremag.com/-L.cfm?Doc=1085-10/2007, 2007.

[DF98] M. Dam and L.-Å. Fredlund. On the verification of open distributed systems. In *SAC '98: Proc. of the 1998 ACM symp. on Applied Comp.*, p. 532–540, New York, NY, USA, 1998. ACM Press.

[DFG98] M. Dam, L.-Å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *COMPOS'97: Revised Lectures from the Int. Symp. on Compositionality: The Significant Difference*, p. 150–185, London, UK, 1998. Springer-Verlag.

[DGRV00] M. Devillers, D. Griffioen, J. Romijn, and F. Vaandrager. Verification of a leader election protocol: Formal methods applied to IEEE 1394. *Formal Methods in System Design*, 16(3):307–320, 2000.

[DIM97] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Trans. Parallel Distrib. Syst.*, 8(4):424–440, 1997.

[DJK+99] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, and B.M. Horowitz. Model-based testing in practice. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, p. 285–294, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[DKR82] D. Dolev, M. Klawe, and M. Rodeh. An $\mathcal{O}(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *J. Algorithms*, 3(3):245–260, 1982.

[EFD05] C. B. Earle, L.-Å. Fredlund, and J. Derrick. Verifying fault tolerant Erlang programs. In *ERLANG '05: Proc. of the 2005 ACM SIGPLAN workshop on Erlang*, p. 26–34, New York, NY, USA, 2005. ACM Press.

[Eri06] Ericsson. Erlang goes multi-core. World Wide Web electronic publication, http://www.ericsson.com/technology/opensource/erlang/-news/archive/erlang_goes_multi_core.shtml, 2006.

[Erl07] *Proc. of the Fifth ACM SIGPLAN Erlang Workshop*, 2007.

[FB97] G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, 1997.

[FGK97] L.-Å. Fredlund, J.F. Groote, and H. Korver. Formal verification of a leader elction protocol in process algebra. *Theoretical Comp. Science*, 177(2):459–486, 1997.

[FGN01] L.-Å. Fredlund, D. Gurov, and T. Noll. Semi-automated verification of Erlang code. In *ASE '01: Proc. of the 16th IEEE Int. Conf. on Automated Softw. Eng.*, p. 319, Washington, DC, USA, 2001. IEEE Computer Society.

[FGN+03] L.-Å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *Int. J. on Softw. Tools for Technol. Transf.*, 4(4):405–420, August 2003.

[Fra82] R. Franklin. On an improved algorithm for decentralized extrema finding in circular configurations of processors. *Commun. ACM*, 25(5):336–337, 1982.

[Fre01] L.-Å. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

[FS07] L-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. In *Proc. of Int. Conf. on Functional Programming (ICFP)*. ACM SIGPLAN, 2007.

[FTW06] L. Frantzen, J. Tretmans, and T.A.C. Willemse. A symbolic framework for model-based testing. In K. Havelund, M. Nunez, G. Rosu, and B. Wolff, ed., *Formal Approaches to Softw. Testing and Runtime Verification - FATES/RV'06*, vol. 4262 of *LNCS*, p. 40–54. Springer-Verlag, 2006.

[GM82] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. on Comp.*, C-31(1):48–59, January 1982.

[GM96] H. Garavel and L. Mounier. Specification and verification of various distributed leader election algorithms for unidirectional ring networks. *Science of Comp. Programming*, 29(1-2):171–197, 1996.

[Ham94] R. Hamlet. Random testing. In J.Marciniak, ed., *Encyclopedia of Software Engineering*, p. 970–978. Wiley, 1994.

[Har99] J. Harrison. A machine-checked theory of floating point arithmetic. In *TPHOLs '99: Proc. of the 12th Int. Conf. on Theorem Proving in Higher Order Logics*, p. 113–130, London, UK, 1999. Springer-Verlag.

[Har01] D. Harel. From play-in scenarios to code: An achievable dream. *Comp. J.*, 34(1):53–60, 2001.

[HBUP03] H. Hallal, S. Boroday, A. Ulrich, and A. Petrenko. An automata-based approach to property testing in event traces. In *Testing of Commun. Systems*, vol. 2644 of *LNCS*, p. 180–196. Springer Berlin / Heidelberg, 2003.

[HJ00] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In *FASE '00: Proc. of the Third Int. Conf. on Fundamental Approaches to Softw. Eng.*, p. 284–303, London, UK, 2000. Springer-Verlag.

[HM85] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *J. ACM*, 32(1):137–161, 1985.

[Hoa62] C.A.R. Hoare. Quicksort. *The Comp. J.*, 5(1):10–16, 1962.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hol97] G.J. Holzmann. The model checker SPIN. *Softw. Eng.*, 23(5):279–295, 1997.

[Hop81] G.M. Hopper. The first bug. *Annals of the History of Comp.*, 3(3):285–286, July–Sept. 1981.

[HR00] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press, 2nd edition, 2000.

[HS02] G.J. Holzmann and M.H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. Softw. Eng.*, 28(4):364–377, 2002.

[Hua93] S.-T. Huang. Leader election in uniform rings. *ACM Trans. Program. Lang. Syst.*, 15(3):563–573, 1993.

[Huc99] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *ICFP '99: Proc. of the Fourth ACM SIGPLAN Int. Conf. on Functional Programming*, p. 261–272, New York, NY, USA, 1999. ACM Press.

[Huc01] F. Huch. Model checking erlang programs - abstracting the context-free structure. In *Workshop on Softw. Model Checking (in conn. w. CAV'01)*, vol. 55 of *ENTCS*, p. 304–321, 2001.

[Huc02] F. Huch. Model checking erlang programs - abstracting recursive function calls. In *Int. Workshop on Functional and (Constraint) Logic Prog. Selected Papers*, vol. 64 of *ENTCS*, p. 195–219, 2002.

[Hug07] J. Hughes. QuickCheck testing for fun and profit. In Michael Hanus, ed., *Practical Aspects of Declarative Languages*, vol. 4354 of *LNCS*, p. 1–32. Springer-Verlag, Berlin Heidelberg, 2007.

[Ins02] Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. Study commissioned by the Department of Commerce's National Institute of Standards and Technology (NIST), 2002.

[IR90] A. Itai and M. Rodeh. Symmetry breaking in distributed networks. *Inf. Comput.*, 88(1):60–87, 1990.

[KM08] M. Kaufmann and J.S. Moore. ACL2 homepage. World Wide Web electronic publication, 2008.

[Knu77] D.E. Knuth. Notes on the van Emde Boas construction of priority queues: An instructive use of recursion. Informally distributed memo, March 1977.

[LeL77] G. LeLann. Distributed systems – towards a formal approach. In B. Gilchrist, ed., *Information Processing 77*, p. 155–160. North-Holland, 1977.

[Lyu95] M.R. Lyu, ed. *Software Fault Tolerance*. John Wiley & Sons, 1995.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[MH89] C.E. McDowell and D.P. Helmbold. Debugging concurrent programs. *ACM Comp. Surv.*, 21(4):593–622, 1989.

[Mod] Modex. http://cm.bell-labs.com/cm/cs/what/modex/index.html.

[Moo03] J.S. Moore. Proving theorems about java and the jvm with acl2. In M. Broy and M. Pizka, ed., *Models, Algebras and Logic of Eng. Softw.*, p. 227–290. IOS Press, Amsterdam, 2003.

[Mye79] G.J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979.

[Nol01] T. Noll. A rewriting logic implementation of Erlang. In M. v.d. Brand and D. Parigot, ed., *Proc. of First Workshop on Lang. Desc. Tools and Applications (ETAPS/LDTA'01)*, vol. 44 of *Electronic Notes in Theoretical Comp. Science*. Elsevier Science Publishers, 2001.

[NPW02] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.

[NR05] T. Noll and C.K. Roy. Modeling erlang in the pi-calculus. In *Proc. of the ACM SIGPLAN 2005 Erlang Workshop*, p. 72–77. ACM, 2005.

[OA99] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In R. France and B. Rumpe, ed., *UML'99 - The Unified Modeling Language. Beyond the Standard. Second Int. Conf. Proc.*, vol. 1723, p. 416–429. Springer, 1999.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: a prototype verification system. In *CADE-11: Proc. of the 11th Int. Conf. on Automated Deduction*, p. 748–752, London, UK, 1992. Springer-Verlag.

[Pel98] D. Peled. Ten years of partial order reduction. In *CAV '98: Proc. of the 10th Int. Conf. on Comp. Aided Verification*, p. 17–28, London, UK, 1998. Springer-Verlag.

[Pet82] G.L. Peterson. An $\mathcal{O}(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Trans. Program. Lang. Syst.*, 4(4):758–762, 1982.

[PST91] B. Potter, J. Sinclair, and D. Till. *An introduction to formal specification and Z.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[Rom01] J. Romijn. A timed verification of the IEEE 1394 leader election protocol. *Formal Methods in System Design*, 19(2):165–194, 2001. special issue of FMICS 1999.

[RV01] J.A. Robinson and A. Voronkov, ed. *Handbook of Automated Reasoning (in 2 volumes).* Elsevier and MIT Press, 2001.

[SC96] P. Stocks and D. Carrington. A framework for specification-based testing. *IEEE Trans. Softw. Eng.*, 22(11):777–793, 1996.

[SF07] H. Svensson and L.-Å. Fredlund. A more accurate semantics for distributed Erlang. In *Erlang '07: Proc. of the 2007 SIGPLAN Erlang Workshop*, p. 43–54, New York, NY, USA, 2007. ACM.

[Sin96] G. Singh. Leader election in the presence of link failures. In *IEEE Trans. on Parallel and Dist. Syst., Vol 7.* IEEE computer society, 1996.

[Som06] I. Sommerville. *Software Engineering.* Pearson Education, 8th edition, 2006.

[Sto97] S.D. Stoller. Leader election in distributed systems with crash failures. Technical Report 481, Computer Science Dept., Indiana University, May 1997. Revised July 1997.

[Sto00] S.D. Stoller. Leader election in asynchronous distributed systems. *IEEE Trans. on Comp.*, 49(3):283–284, March 2000.

[Tan96] A. Tanenbaum. *Computer networks (3rd ed.).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[TB99] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7^{th} European Int. Conf. on Softw. Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.

[Tel00] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2000.

[Tor97] S. Torstendahl. Open telecom platform. *Ericsson Review*, no. 01/1997.

[TP00] W. Torres-Pomales. Software fault tolerance: A tutorial. Technical report, NASA Langley Technical Report Server, 2000.

[Tre92] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[Use99] Y.S. Usenko. A comparison of spin and the muCRL toolset on HAVi leader election protocol. In *279*, p. 23. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X, 31 1999.

[Val98] A. Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, p. 429–528, London, UK, 1998. Springer-Verlag.

[Wei81] M. Weiser. Program slicing. In *ICSE '81: Proc. of the Fifth Int. Conf. on Softw. Eng.*, p. 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[Wid04] M. Widera. Flow graphs for testing sequential Erlang programs. In *ERLANG '04: Proc. of the 2004 ACM SIGPLAN workshop on Erlang*, p. 48–53, New York, NY, USA, 2004. ACM Press.

[Wig03] U. Wiger. Fault tolerant leader election, 2003. http://www.erlang.org/.

[Wik94] C. Wikström. Distributed programming in Erlang. In *PASCO'94, First Int. Symp. on Parallel Symbolic Computation*, Linz, Austria, Dec 1994.

Paper 1

Semi-Formal Development of a Fault-Tolerant Leader Election Protocol in Erlang

This paper was written together with Thomas Arts and Koen Claessen. The paper was published at the Workshop 'Formal Approaches to Testing of Software' (FATES), in Linz, Austria, September 2004. The paper included here is a slightly updated and reformatted version.

Semi-Formal Development of a Fault-Tolerant Leader Election Protocol in Erlang

Thomas Arts[1], Koen Claessen[2], and Hans Svensson[2]

[1] IT University in Göteborg, Box 8718, 402 75 Göteborg, Sweden
thomas.arts@ituniv.se

[2] Chalmers University of Technology, Göteborg, Sweden
{koen,hanssv}@cs.chalmers.se

Abstract

We present a semi-formal analysis method for fault-tolerant distributed algorithms written in the distributed functional programming language Erlang. In this setting, standard model checking techniques are often too expensive or too limiting, whereas testing techniques often do not cover enough of the state space.

Our idea is to first run instances of the algorithm on generated stimuli, thereby creating *traces* of events and states. Then, using an abstraction function specified by the user, our tool generates from these traces an abstract state transition diagram of the system, which can be nicely visualized and thus greatly helps in debugging the system. Lastly, formal requirements of the system specified in temporal logic can be checked automatically to hold for the generated abstract state transition diagram. Because the state transition diagram is abstract, we know that the checked requirements hold for a lot more traces than just the traces we actually ran.

We have applied our method to a commonly used open-source fault-tolerant leader election algorithm, and discovered two serious bugs. We have also implemented a new algorithm that does not have these bugs.

1 Introduction

The company Ericsson has developed a telecommunication switch called the AXD 301 [7]. The control software of this switch is written in the distributed functional programming language Erlang [2]. A major challenge in the development of the switching software is to get the almost one million lines of code tested in the relatively short time between releases of the product. A typical time consuming and difficult activity is testing fault-tolerance properties. The particular fault-tolerance we investigate here is the effect of taking down parts of a switch (because of maintenance or hardware problems) and restarting them later in time.

We report on our case study to take away part of the testing load by analyzing a critical part of the code by semi-formal methods. The part we looked at is a *leader election* protocol. In the AXD 301, a module of about 2000 lines of code implements both a leader election protocol and a resource manager. In order to be able to deal with the complexity of this module, Ericsson's engineers rewrote the module in two parts, separating the resource manager and the leader election protocol. The simplified resource manager has been formally verified in earlier work by using a model checking approach [3]. The slightly generalized and cleaned

up leader election protocol contains about 800 lines of code and is available as open source [21].

The leader election problem is a well-known and extensively studied problem. The objective of the protocol is for the processes among themselves to establish a designated process, called the *leader*. Leader election protocols have been designed for many different settings. In our case, we are interested in a solution that is fault-tolerant. Fault-tolerance is based on communication links breaking or on processes that may die or revive again at any point in time. If the currently elected leader dies or is disconnected, the surviving processes need to elect a new leader amongst them. However, during the election process, other processes may cease to work. We consider asynchronous communication with buffered messages. Among the many articles published on the leader election protocol [9, 6, 16, 17, 1], we know of only a few that address all these problems. It was actually hard to find a paper describing exactly the setting that Erlang uses: asynchronous message passing, reliable communication channels between every pair of processes, possible failure and/or revival of a process at any point in time and a reliable notification mechanism of when processes die.

The algorithm used in the leader election protocol implementation we analyzed was an adaptation of a previously published algorithm [16]. There is a fixed set of processes that can die arbitrarily, and they have to negotiate a leader among them. The first process that comes up has priority to become leader, in order to have a selected leader as soon as possible. Only when the current leader dies should a new leader be elected.

There are two basic properties that the leader election implementation needs to obey: (1) **Safety** — it is never the case that there are two or more leaders at the same time; (2) **Liveness** — in a stable situation (i.e. processes stop dying for a while), a leader will eventually be elected.

We have considered using model checking techniques to formally verify these properties. However, we found that dealing with the fault-tolerance leads to state-space explosion in the model checkers used, which severely limited the number of processes we could deal with. More informal methods based on testing seemed to be necessary.

The Erlang runtime system has built-in support for generating *traces* of the events occurring during execution. With simple means, one can specify what one considers an event (sending a message, receiving a message, a process dying, a function call, etc.). Tracing can be switched on and off on demand. Studying the traces reveals not only that an error occurred, but can also demonstrate the chain of events that led to the error.

We have developed a methodology for semi-formal analysis of such traces of distributed systems (c.f. [5]). The idea is to first produce traces of the system by generating stimuli, as in testing. Then, we build *abstractions* of the traces with the help of an *abstraction function* specified by the user. An abstraction function basically maps data structures in the events and states to different (simpler) representations. An abstraction reduces the number of states, by mapping the actual concrete states onto a set of abstract states. This also allows us to detect cyclic behaviour in the trace, since different concrete states can be mapped to the

same abstract state. The accompanying abstract state transition diagram concisely indicates the different abstract states visited during an execution, together with the messages sent and received during the transitions. A path through such a state diagram represents a trace, but it is not necessarily the case that the trace is a possible trace for the system since the diagram is really a diagram for an abstracted model of the system.

We propose to generate traces of simple instances of the software first, such as a reduced number of processes or executing only one possible scenario. For these traces it is easy to define abstraction functions. The same functions can be used for more complicated instances of the software.

The generated abstractions are used in two ways: (1) They increase understanding of the system, and can help to more easily spot the causes of bugs (as explained in Section 2.4); (2) We can formally verify properties of the abstraction, thus ensuring that the desired properties actually hold for *all* paths through the abstract state diagram (as explained in Section 2.7).

When we applied our methodology on the implementation of the leader election algorithm, we discovered two serious bugs. Failing to correct the bugs in an efficient way, we also tried to implement a different algorithm for leader election. This implementation is based on [17] and was tested with the methodology described in this paper without finding any errors.

2 Methodology

In this section, we describe our methodology in more detail. We do this by concretely following the analysis of a leader election algorithm in chronological order. We start by describing the original algorithm, how we generate stimuli to obtain system traces, and how we use abstractions to find bugs. Then, we describe our own implementation of leader election, where all of our analyses failed to find any errors, and discuss coverage issues related to our method.

2.1 Fault-Tolerant Leader Election Version 1

The Erlang code for the algorithm we started with is publicly available on the web [21]. However, for simplification purposes we actually analyze a cut-down version of this code here. All code we used in this case study is available on the web [18].

The implementation is loosely based on a fault-tolerant leader election algorithm described in [16], but with an adaptation in order to deal with faults being dying processes instead of failing communication links.

The participating processes behave as follows. When the protocol is started each process is given a list of all the participation processes; the position in this list is also the priority order for the processes. A process always plays one of the following four roles: *candidate*, *captured*, *surrendered* or *elected*. When a process is started it is always a candidate to become a leader. The first thing it does is to try to capture all the other processes, by broadcasting a 'capture'-message. If another candidate-process receives a 'capture'-message, the receiving process will

take action based on the priority; it will ignore messages from processes with lower priority, and accept messages from processes with higher priority by sending an 'accept'-message. After accepting, the process changes its role to being captured. A captured process will ignore 'capture'-messages and forward 'accept'-messages to the process that has captured it. Whenever a candidate has captured more than half of the participating living processes, it will announce itself as the leader by broadcasting an 'elect'-message. If a process receives an 'elect'-message it will immediately surrender.

Whenever the leader dies (processes discover this since the Erlang runtime system will send a 'DOWN' message to all interested processes), a new election round is started. Whenever a process revives, this process will be notified of the (possible) presence of a leader via an 'elect'-message as a reply from the leader to the 'capture'-message sent when the process revived.

When we got the algorithm, it was said that it would always eventually choose a leader if more than 50% of the processes are alive and if the system is stable for long enough. (It is though possible that a leader is elected with fewer processes alive.) The algorithm is not supposed to elect a new leader unless the leader dies.

2.2 Generating Stimuli and Tracing

The Erlang Runtime System (ERTS) has built in functionality for tracing running processes. The tracing can be switched on or off at any given time, without interfering with the execution. It is possible to trace sent and received messages, function calls, process related events, process scheduling and garbage collection. In a distributed environment there exists functionality for redirecting trace messages to a central collection process, in order to collect all trace data into the same log file.

Stimuli for leader election implementation

In order to generate traces of the leader election protocol, a set of nodes is started, and a leader election process is started on each node. The stimuli for a leader election system are killing and reviving processes. A simulation process then randomly selects which process to kill/revive by sending messages to the control processes. How many processes can be dead at the same time is configured in the simulator process.

In order to further test the robustness of the leader election protocol, we implemented a variant where we also delay messages between nodes in a random way. The idea is that this simulates slow and/or overloaded connections. Note that this is not tested in a standard setting where one runs all nodes on the same hardware, since communication delays will be rather static in such a setting.

Tracing the implementation

We first collected trace data for the simplified version of the leader election protocol, without using message delays. When running a leader election system

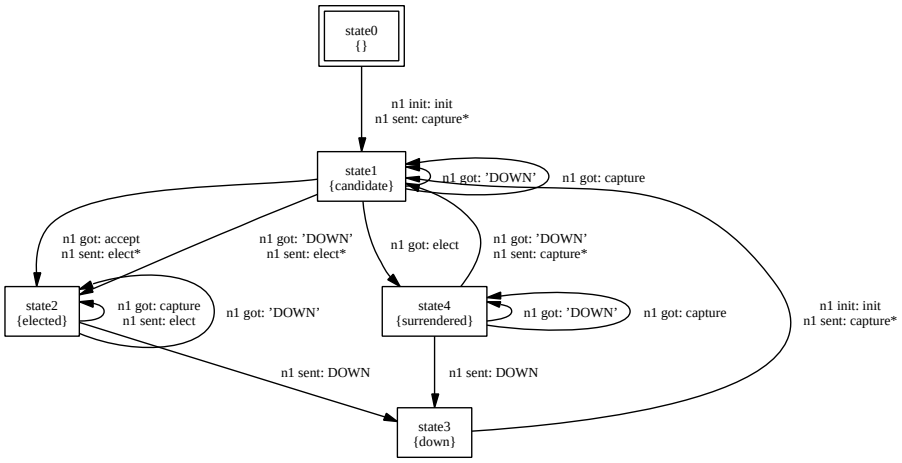


Figure 1: Abstract trace for one process in a three node setting

with three processes, everything worked fine, but when running with five processes something was obviously wrong; there were two processes simultaneously announcing themselves as leader! In the search for this error, we focused on the trace data for one of the nodes which was elected as leader. The raw trace data contained roughly 120 states and 200 message events, a bit too many for easy overview. The problem here is that it is easy to spot where in the trace the fault happened (two leaders are elected), but not where in the trace the event happened that triggered the fault (the first illegal state).

2.3 Abstractions

It is clear that in order to understand larger traces of systems, one has to reduce the information in the trace to a relevant subset of all information. One way of doing this is by using an *abstraction* (c.f. [5]). Abstractions are made by applying an *abstraction function* that converts each concrete state in the trace to an abstract state, which contains less information. Several different concrete states from the trace might actually be mapped to the same abstract state. Thus, we can redisplay the trace by means of a state transition diagram, where each abstract state occurs only once, and transitions occur between two abstract states if there exists a transition in the trace between two corresponding concrete states. However, by doing this we also lose some context, for example a state visited exactly N times in the actual trace is represented by a loop, and thus potentially infinitely visits, in the abstract trace. Moreover, we can also make the sent and received messages more abstract by applying a message abstraction function.

An example of an abstract state transition diagram of the leader election protocol is displayed in Fig. 1. Our tool automatically generates this diagram, given an abstraction function specified by the user. The original trace used for this diagram is a trace of a correct execution with three leader election processes.

Here, the abstraction function on states is tracking the state of only one process, abstracting away the states of the other processes. Moreover, it has removed all other information in the concrete state, but for the role a process is playing. This diagram shows that with help of an abstraction, one can get an understanding for the basic parts of the algorithm, since it is easy to follow how the process moves between the different roles. We call the transition diagram generated from a trace and an abstraction function an *abstract trace*.

Common abstraction function building blocks

We have implemented a library of common abstraction function building blocks. Commonly used functions are: removing parts of state data, replacing a list by its length, focusing on the state of one process, merging states of two processes into one state, etc. This library makes it easy to quickly define new abstraction functions.

2.4 Abstractions for Bug Finding

The idea is now to find an abstraction function which clearly helps us to establish where in the code the bug is located. It is hard to give a general approach on how to come up with an appropriate abstraction. Most of the time, the programmer has some sort of intuition about what parts of the states and which events influence a particular bug.

In the case of our bug, we have applied the following principles. Some of the state data, such as the list of participating nodes, is the same in all states, and such data can often be abstracted from. In the state data there are also two lists, containing the references to monitored nodes and the nodes which are down. The contents of these lists are not really useful, it is enough to know how many elements there are in the lists. So, we abstract away from these lists by remembering their length, but not their content. Concerning the events, most of the message data can be abstracted away, only keeping the type of a message.

The above abstraction reduces the state space from 120 to 23 states, which is small enough to overview. It is now possible to spot the bug by just looking at the abstract trace (Fig. 2). The state where the process is elected as leader is dark shaded in the figure (it is in the lower left half). This state is part of a long almost non-forking path, and it is likely that the first illegal state is to find at the top of this path. This is indeed the case and if we zoom in on two lightly shaded states in the upper left part of Fig. 2 the result can be seen in Fig. 3.

Let us examine more closely what happens in the bug-containing trace. The state data contains three fields: the role, the number of nodes that are down, and the number of monitored nodes. In the state labeled 'state9' we can see that the list of dead processes contain one process, and the list of monitored processes contain four processes. Since processes should not monitor themselves, this is clearly one process too many.

This bug turned out to be a mistake we made ourselves, when implementing the cut-down version of the algorithm. We had not been careful enough in the

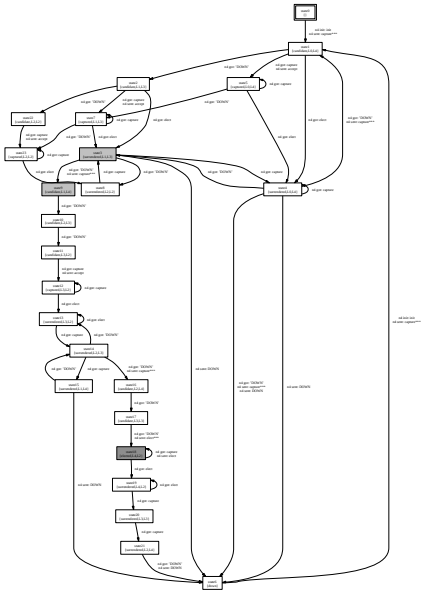


Figure 2: Abstract trace containing bug

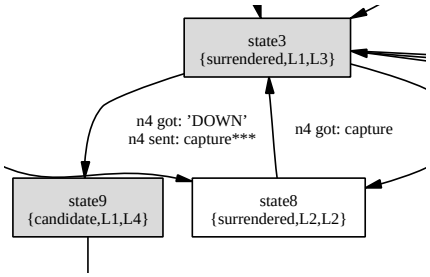


Figure 3: Faulty part of abstract trace

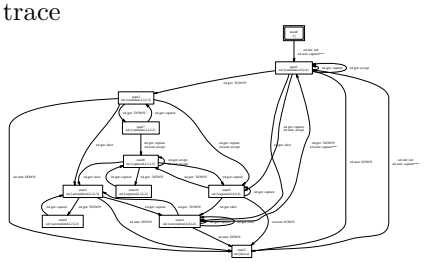


Figure 4: Bug-free abstract trace

implementation and mixed up variable names. It shows, however, the usefulness and simplicity of the approach.

If we compare the abstract trace in Fig. 2 where the bug is present with an abstraction made from a trace where the bug is fixed in Fig. 4, one can clearly see from the graph structure that the erroneous behaviour is gone.

The first serious bug

After correcting this bug, we collected a new set of traces. This time we initially observed no obvious faulty behaviour, we therefore activated the random delaying of messages in the generation of stimuli. Now we could observe a faulty behaviour, this time in a leader election system with only three nodes, and again it was a violation of the safety property: Two nodes simultaneously announced themselves as leaders. Again we turned to abstract traces in the search for an explanation. In this case we found the error to be present in situations where many nodes failed simultaneously.

Consider the situation in Fig. 5, where initially only process A is alive and the priority of the processes is A > B > C. If then B and C revive more or less simultaneously and the present leader (A) is suffering from slow connections, it is possible that the newly revived processes will agree on a leader before the present leader is able to announce its presence.

This is indeed a serious bug, and this bug is present also in the original Erlang code. But it is also the case that this situation will not occur if the system is

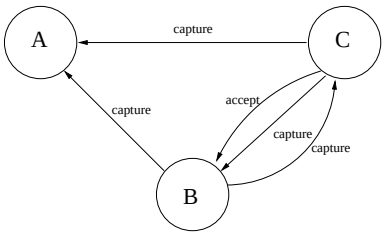


Figure 5: Deadlock situation in leader election protocol

simulated in such a way as to always have more than half of the processes alive. So, we continued the analysis of the protocol with less aggressive stimuli.

2.5 Sanity Checks on Abstractions

We call an abstract trace *sufficient* if all real traces of the system are embedded in it. Note that by construction, it is guaranteed that at least the original trace is embedded in the abstract trace. If all possible traces of the system are embedded, we cover all possible executions of the system. If an abstract trace is sufficient and a property holds for this abstract trace, then it also holds for all real traces. However, in general we do not know whether an abstract trace is sufficient. This is related to coverage and is discussed in Sect. 2.8.

No quiescent states

There are other problematic states where the system can get stuck. Remember that we stimulate the system by taking down and reviving processes arbitrarily during tracing. If there exist a state in an abstract trace that has only one outgoing arc labeled with a 'DOWN'-message of a process, something is wrong as well. This means that the system is in a state where the only way to get out is for a process to die. Since there are no guarantee that processes eventually will die, the system is stuck in that state.

There might be two reasons for this. One is that the abstract trace is insufficient (which means that we should have chosen a different abstraction function, or collected more trace data). The other is that the system has a deadlock in that state (which could indicate an error). Our tool automatically reports such quiescent states.

The second serious bug

When we investigate quiescent states for our leader election algorithm, there is a warning for some potential deadlock nodes. Most of those can immediately be discarded, since these are states in which there is a leader elected and hence are not problematic states.

But there is indeed a quiescent state which indicates a real deadlock! In some cases when the leader process dies, the remaining processes end up in a state

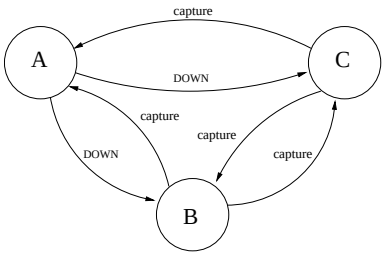


Figure 6: Deadlock situation in leader election protocol

where a process is waiting for a message that is not going to be sent. Consider the situation in Fig. 6, where all the processes are initially alive, A is the leader and the priority of the processes is A > B > C. Then if A is killed, B and C are notified of this and each receive a 'DOWN'-message. Now, if the message to B is faster than the message to C, it is possible for B to start a new election round and send a 'capture'-message to C before C receives the original 'DOWN'-message. In that case C will simply ignore the 'capture'-message, since C (falsely) thinks that A is alive and will answer the 'capture'-message on behalf of C. When C finally gets the 'DOWN'-message, and starts its new election round B will ignore the 'capture'-message from C with the motivation that B is higher prioritized than C, which means that C should reply to B's 'capture'-message instead. Therefore we end up in a situation where B is waiting for a message that C is not going to send. This deadlock situation is not broken until another node dies or revives.

Thus we have discovered yet another bug in the leader election algorithm, this bug is also present in the original, non-simplified, implementation! The error would probably never occur when all nodes run on similar hardware, however, our addition of delays in messages reveals a very tricky error that may show up in very rare circumstances or when the protocol is used with nodes on different hardware.

2.6 Fault-Tolerant Leader Election Version 2

At this point, we had discovered two serious bugs in the original leader election implementation. We were unable to repair the implementation. So, we decided to try to implement and analyze another algorithm for leader election. Our new algorithm is based on 'The Bully Algorithm for Synchronous Systems' in [17], but again we were forced to make some modifications in order to adapt the algorithm to our setting.

The algorithm is quite simple and it is easy to understand how it works. When a process comes up, it first checks whether any process of higher priority is alive. If there is, it waits for one of these processes to become leader. If not, the process itself decides to try to become leader. It then checks that all other processes of lower priority either are aware of its existence, or are dead. If so, it announces itself as leader.

The main change we made to the algorithm in the paper was to avoid restart-

ing the election process each time a process revives. This is inefficient and not applicable to the situation where our leader election protocol is supposed to be used. We made the change in two steps, first we changed the algorithm such that no new election would be started if a process with lower priority than the leader revived and later we took care of the situation where a process with higher priority than the leader revived. This second change was surprisingly complex. We also made some changes that did not affect the functionality, but which reduced the number of messages sent by the system. The code is available on the web [18].

After making the changes, we collected a new set of traces. We created some different abstractions, under which the system seems to be working correctly.

2.7 Abstractions for Verification

So far, we have been able to spot errors exhibited by our abstractions either visually or by means of simple sanity checks. However, when the abstractions or desired properties get more complicated, to be sure that an abstract trace obeys a given property, an automated technique is needed. Our idea is to simply formally check properties of the abstract traces using a model checker.

LTL properties

We formulate the properties that we want to verify in linear time logic (LTL). In the introduction we mentioned two basic properties for a leader election protocol: (1) There are never two elected leaders at the same time; (2) If the system is stable, eventually a leader will be chosen. For a leader election situation with 3 nodes, the first property can be expressed in LTL as follows:

$$\Box(\neg((l_1 \wedge l_2) \vee (l_1 \wedge l_3) \vee (l_2 \wedge l_3))). \tag{1}$$

Here, l_i is defined to be true exactly when the leader election process running on node i is the elected leader. So, the property can be read as: "It is never the case that node 1 and 2 are leader at the same time, or node 1 and 3, or node 2 and 3."

The second property can be expressed as follows:

$$\Box(\Box(\neg l_1 \wedge \neg l_2 \wedge \neg l_3) \Rightarrow \Box\Diamond(d_1 \vee d_2 \vee d_3)). \tag{2}$$

Here, l_i is defined as above, and d_i is true exactly when node i dies. This property can be read as: "The only traces where no leader is chosen are those traces where process die infinitely often."

Checking if the above properties hold for a given abstract trace is done using standard LTL model checking techniques [10].

Improper cycles

Since we are modeling asynchronous message passing using transition systems, the information needed to represent the real state of our system consists of more

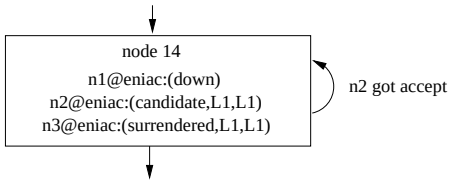


Figure 7: Loop which is not a possible trace.

than simply the state of the transition diagram. We also need to know what messages have been sent that have not arrived yet. This problem is illustrated by some counter examples we get of our properties. A cycle in a counter example that contains a message M that is being received by a transition on the cycle, but not sent by a transition of the cycle can of course never represent a real run of the system. We call such a cycle an *improper cycle*.

An example is displayed in Fig. 7, which displays a situation that can not correspond to an actual trace, since such a trace only consumes 'accept'-messages. This could not be an infinite chain of events, since that would mean that an infinite number of 'accept'-messages has to be produced. So, when we search for counter examples in the LTL model checking algorithm, we also have to check that found cycles contain the production of all messages that are consumed. Our property checker automatically rejects runs that contain such improper cycles.

Results

We have checked that both properties 1 and 2 hold for abstract traces of our new implementation of the leader election algorithm, for up to N processes. We have done most of the testing with $N = 3$ and $N = 5$, but has also used larger N ($N = 7$ and $N = 10$). Note that this does not mean that we have formally verified the above properties for the system; only that all generated abstract traces satisfied the properties.

2.8 Coverage

When discussing test-based methods, the issue of coverage is central. Coverage methods should provide some sort of measure of how much of the system one has exercised, and this is important for evaluating the result of the testing. In general, coverage methods can warn of potential situations where we have *not* tested enough; very seldom we can know that we have indeed tested enough. Therefore, it is good practice to apply as many different coverage measures as possible.

Code coverage

Erlang has a built-in module, `cover`, for various basic kinds of coverage analysis. It is a very standardized set of tools, which basically provides information of how many times each executable line of code has been accessed. The limitations of

point-coverage are well-known. For our new leader election algorithm, we have traced the system such that we exercised all lines of the code that were supposed to be run.

Abstract trace coverage

Instead of looking at how the actual generated traces have exercised the different parts of the system, we can investigate coverage properties of the abstract traces.

A simple way of doing this is to specify quantitative properties of expected events in the abstract traces. For example, for each node, how many states exist where that node has been elected as leader? For each state and each process, how often is it possible to reach a state where that process is dead? How much of the theoretically reachable state space is actually reached?

For our new leader election algorithm, coverage results are of course affected by how much tracing is done, and how the stimuli are chosen. It is interesting to study what will happen with coverage numbers for different amount of tracing. The measures that we considered here are the percentage of reached states and the percentage of the states which could be left via a 'DOWN'-transition. The results are not very surprising, the number of reached states as well as the number of nodes with an outgoing 'DOWN'-transition increased with the length of the traces. The numbers are quickly rising for small amounts of tracing, but levels out after further tracing. In the longest traces we reached 87% of the states in the complete state space (it is not entirely clear that all of the states could indeed be reached). About 30% of states had an outgoing 'DOWN'-transition, a somewhat low number. This could be explained by the fact that the stimuli system was not fast enough, so in many situations a killing could not happen with our simulation technique. It is possible that this can be improved with better stimuli generation.

3 Related Work

The leader election protocol has been extensively studied. There are many variations of this algorithm with different assumptions about the network topology and other constraints. Published leader election algorithms are often proved correct on paper, but implementations tend to divert a bit from the actual algorithm, after which correctness is no longer guaranteed. This happened for example with both implementations we studied, which were based on published algorithms [16, 17].

Formal verification and formal testing are supplementary techniques. We deal with real code, whereas there have been other approaches to deal with models of leader election algorithms. For example, the formal verification of the IEEE 1394 leader election protocol [14] has results that cannot directly be applied to our leader election protocol, since different assumptions are made on the network topology and detection of faults.

The two other model checking approaches that we are aware of [11, 12] deal with algorithms that have constraints that differ from our case. Model checking is possible because the algorithms that are verified are essentially less complex than the one we consider.

Different from formal testing, we do not have a formal model of the software to generate test cases (e.g. [8, 19, 20]). We more or less construct an incomplete model from the real traces. This model is on one hand shown to the engineers for visual verification and on the other hand input to our model checking approach. Given that we call all our traced events observable, we obtain an abstraction in which all real traces are observable in the abstract trace, however, not vice versa. We use executions of the software to obtain a model for the software with a good coverage and apply model checking techniques on the model to test the software.

Compared to the initial work on trace analysis for Erlang [5], we went further than visualizing the traces as graphs, but we actually performed model checking on those graphs. We improved the trace collection mechanism to simulate delays in communication and to be able to handle events that occur quickly after that tracing starts (events that we missed in the earlier setting). The latter was necessary to be able to deal with re-starting processes.

Another project working with trace analysis is the Java PathExplorer [13]. With this tool it is possible to specify properties for Java programs in temporal logic. The program is instrumented to emit events when executed. The properties are then checked for the event stream. The related tool Java MultiPathExplorer [15] takes the concept a bit further by also being able to generate more possible traces from a single observed trace. This is done by reordering of unrelated events. This technique could be complementary to our method that uses abstractions to generate more possible traces.

4 Conclusions and Future Work

In this paper we describe a case-study in which we use abstraction of traces to analyze a complex software component. By using this technique, we were able to identify two errors in the code. We re-designed the code and verified it by the same technique of trace abstraction, not finding any errors this time.

The described methodology of analysis and abstraction of traces is generally applicable to Erlang programs, in particular to the kind of software that is written in industrial projects. The primitives necessary to create a trace are part of the standard Erlang runtime system. Generating traces is rather common testing technology for engineers working with Erlang software. However, so far, engineers look at the output traces as a textual long list of events. By the possibility of visual verification, i.e. inspection of the graphs obtained from an abstracted trace, motivation is created to write those abstraction functions [5]. Compared to writing extra code for testing Erlang code, writing the abstraction functions really is a minor job, since they only address data conversion of state and messages.

The first thing we achieve by using abstracted traces instead of analyzing the real traces, is that there is less ‘noise’ in the output. With manual inspection of a trace, it makes a difference whether one looks at 2000 long events or a few dozens of short events. The second advantage is that the abstraction allows us to detect cyclic behaviour, which need not necessarily be cyclic behaviour in the original trace. For example, if one abstracts from a time stamp, one would be able to see a certain message repetitively been sent from a certain state, whereas

with the time stamp, it occurs as non-cyclic in the trace. The additional cycles not only make the trace shorter, but they also give extra insight in the behaviour of the software. Third, one can prove properties over the abstract traces, which then hold for many more than just the original trace. A property proved for an abstract trace holds for all traces that result in the same abstraction. In that way, we achieve a larger coverage by only looking at a few traces.

Since the methodology of generating traces in general cannot guarantee full coverage, we use it for identifying errors instead of proving correctness. By proving properties that should hold, we know that something is wrong if we get a counter example. If we cannot exhibit the found error in the actual trace, we might have used an inadequate abstraction. For example, the abstract state space may contain cycles that do not correspond to a cycle in the real code. Thus, we can detect errors in the code, but we pay the price of possibly seeing some false negatives. However, these false negatives can also result in a better understanding of the system.

As mentioned in the introduction, part of the AXD 301 software was verified by using a model checking approach. Is the same approach applicable here? First of all, the tool to generate the state space of an Erlang program [4] could not be directly applied to the code. The tool abstracts away from process failures, thus we could only verify all runs in which none of the processes died. Here we could confirm that indeed a leader was selected on all branches.

It is ongoing work to add process failure and recovery to the tool. We added it by hand to the model we obtained from the tool, immediately spotting two major problems. First, there is the obvious state space explosion problem, resulting from the explosion in possible events that can happen in different orders. Second, the way message passing is modeled by the tool is too restrictive and excludes particular orders of events that could happen in reality. Thus, with the present available technology, it is a real challenge to verify the properties we are interested in with a model checker. Therefore we think that one should first apply the much cheaper tracing technology to find errors in the code. In case one cannot find any error, it might be beneficial to generate the whole state space and use the same abstraction functions to reduce the model and prove the properties of interest.

Future Work

Possible future work includes automating the creation of the used abstraction functions. We have also considered developing a design document that helps software engineers to quickly create useful abstraction functions.

We would also like to see if it is possible to integrate our abstraction functions with standard model checking techniques based on abstraction. In order to increase the capacity (e.g. number of participating processes) of model checking techniques even more, we probably even need to use symmetry reduction or symbolic model checking.

References

[1] M. Aguilera, C. Delporte-Gallet, and H. Fauconnier. Stable leader election. In *Distributed Computing, 15th International Conference DISC2001*, volume 2180 of *Lecture Notes in Computer Science*. Springer-Verlag, October 2001.

[2] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1996.

[3] T. Arts, C. Benac Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *Int. J. on Software Tools for Technology Transfer*, 2004. to appear.

[4] T. Arts, C. Benac Earle, and J. J. Sánchez Penas. Translating Erlang to mCRL. In *Fourth International Conference on Application of Concurrency to System Design*, Hamilton (Ontario), Canada, June 2004. IEEE computer society.

[5] T. Arts and L.-Å. Fredlund. Trace analysis of Erlang programs. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 16–23. ACM Press, 2002.

[6] N. Bjørner, U. Lerner, and Z. Manna. Deductive verification of parameterized fault-tolerant systems: A case study. In *Proceedings of the 2nd International Conference on Temporal Logic*. Kluwer, 1997.

[7] S. Blau and J. Rooth. AXD 301 - A new generation ATM switching system. *Ericsson Review*, 1:10–17, 1998.

[8] E. Brinksma. A theory for the derivation of tests. *Protocol Specification, Testing and Verification*, VIII:63–74, 1988.

[9] J. Brunekreef and S. M. J.-P. Katoen, R. Koymans. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, 9(4):157–171, 1996.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[11] L.-Å. Fredlund, J. Groote, and H. Korver. Formal verification of a leader elction protocol in process algebra. *Theoretical Computer Science*, 177(2):459–486, 1997.

[12] H. Garavel and L. Mounier. Specification and verification of various distributed leader election algorithms for unidirectional ring networks. *Science of Computer Programming*, 29(1-2):171–197, 1996.

[13] K. Havelund and G. R. su. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189 – 215, March 2004.

[14] J. Romijn. A timed verification of the IEEE 1394 leader election protocol. *Formal Methods in System Design*, 19(2):165–194, 2001. special issue of FMICS 1999.

[15] K. Sen, G. R. su, and G. Agha. Runtime safety analysis of multithreaded programs. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 337–346. ACM Press, 2003.

[16] G. Singh. Leader election in the presence of link failures. In *IEEE Transactions on Parallel and Distributed Systems, Vol 7*. IEEE computer society, 1996.

[17] S. Stoller. Leader election in distributed systems with crash failures. Technical Report 481, Computer Science Dept., Indiana University, May 1997. Revised July 1997.

[18] H. Svensson. Various material related to the paper, including examples. http://www.cs.chalmers.se/∼hanssv/erlang_testing.

[19] J. Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[20] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.

[21] U. Wiger. Fault tolerant leader election. http://www.erlang.org/.

Paper 2

A New Leader Election Implementation

This paper was written together with Thomas Arts. The paper was published at the 4th 'ACM SIGPLAN Erlang Workshop' in Tallinn, Estonia, September 2005. The paper included here has a few minor corrections and is also typeset in a different style.

A New Leader Election Implementation*

Hans Svensson[1], and Thomas Arts[2]

[1] Chalmers University of Technology, Göteborg, Sweden
hanssv@cs.chalmers.se

[2] IT University in Göteborg, Box 8718, 402 75 Göteborg, Sweden
thomas.arts@ituniv.se

Abstract

In this article we introduce a new implementation of a leader election algorithm used in the generic leader behavior known as gen_leader.erl. The first open source release of the generic leader [6] contains a few errors. The new implementation is based on a different algorithm, which has been adopted to fulfill the existing requirements. The testing techniques used to identify the errors in the first implementation have also been used to check the implementation we propose here. We even extended the amount of testing and used an additional new testing technique to increase our confidence in the implementation of this very tricky algorithm. The new implementation passed all tests successfully. In this paper we describe the algorithm and we discuss the testing techniques used during the implementation.

Categories and Subject Descriptors: D.2 [*Software Engineering*]

General Terms: Algorithms, Verification

Keywords: Erlang, leader election, distributed systems, implementation

1 Introduction

Many distributed applications are easy to implement if one has one dedicated process to administer certain tasks. For example, one process could poll all attached hardware devices to determine the configuration of a distributed system, whereafter the other nodes may then consult this process for the configuration information. More generally, it is often useful to have a server process that is in charge of keeping a consistent view of an aspect of the system state. All nodes in the distributed system consult that server process if they want information about the system state or if they want to update the system state.

A dedicated server provides an easy way to introduce consensus, synchronization and resource allocation in a distributed system. The disadvantage with this solution is, though, that one introduces a single point of failure in the system. In a fault-tolerant setting, at least one stand-by node needs to be introduced. Taking that thought one step further, several stand-by nodes may be introduced, since that provides an even better protection against faults. With either one or more stand-by nodes, each stand-by node has the problem of detecting when to become the active node. In fact, the primary node (the one that is assumed to run the dedicated server if nothing goes wrong) also has the problem to determine whether it can actually take that role. This is caused by the fact that when this primary node starts, one of the stand-by nodes may already have decided that the primary node is dead and that it should run the server instead.

This problem of having several nodes competing to perform one central task is well-known and described in literature as *the leader election problem*. A solution to this problem is an algorithm that when its execution terminates, guarantees that a single node is designated as a leader and every node knows whether it is a leader or not. The leader is then assigned the role of the above described dedicated server.

At least since the early seventies leader election algorithms for all kind of settings have been described. Often these solutions are stated in form of a multiprocessor machine with shared memory or by means of computers in a token-ring network. Most solutions assume a perfect world in which no failures occur. Some solutions assume possible failure of the communication others possible failure of the nodes. There are over $10,000$ articles on the leader election problem and it is not easy to find a solution among them that fits the Erlang context well.

In our case, we are interested in a solution that is fault-tolerant with respect to failing and restarting processes and failing and restarting nodes. We assume Erlang nodes to have reliable communication without lost messages (basically the TCP/IP setting in which all nodes can directly communicate with all other nodes in a reliable way). In the open source Erlang community there exists an implementation of a leader election algorithm [6]. This implementation is based on an article written by Singh [4], but contains numerous adaptations to the Erlang setting. The implementation originates from the work at Ericsson with the AXD 301 telecommunication switch, but has been rewritten and turned into the OTP behavior `gen_leader`. From a user point of view, the generic leader behaves like a generic server with callback functions like *call* and *cast*. The intended use is that of having one generic leader per node and clients access only the generic leader on their node. The generic leaders communicate with each other and forward all requests to the chosen leader.

Thorough tests have shown that the above mentioned implementation, unfortunately, contains errors (see [1] for details). In some rare circumstances, two leaders can be elected at the same time. In addition, there is a possibility that the election of a new leader stands in a deadlock. The system may run for years without showing any failure, but there is always the potential danger that one day the circumstances are exactly such that those faults occur.

After failing to repair the implementation we proceeded to make a new im-

plementation based on another algorithm. The new implementation is based on the article 'Leader Election in Distributed Systems with Crash Failures' by Stoller [5]. Compared with Singh, Stoller takes a slightly different approach to the leader election problem, which seems to fit better into the Erlang setting. However, we still had to modify the algorithm, since it was designed for a completely different situation.

We took care to supply the same interface for this new implementation as defined for the original, incorrect, implementation [6]. However, due to the differences in the implemented algorithms the interface functions that return all alive nodes and the one returning all dead nodes, could not be provided. Apart from that the behavior of the new implementation should be, when viewed from the outside, the same as the behavior of the old implementation. Except for the failures!

We have tested the implementation thoroughly, using both the test method with abstract traces that revealed the errors in the original implementation [1], and Erlang QuickCheck [3] which is a property-based random testing tool. While testing the implementation we discovered and successfully corrected a number of errors we made in the implementation. The new version of gen_leader is available at http://www.cs.chalmers.se/~hanssv/leader_election.

In Sect. 2 we explain the algorithm we have implemented, and the adaptations that were made to make the algorithm useful in this context. In Sect. 3 we describe the implementation and the testing of the implementation. We conclude with discussion in Sect. 4.

2 Algorithm

Sometime shortly after the first implementation [6] was written *Google* was used to search for the source of Singh's algorithm [4] (on which the implementation was based). During that search for leader election algorithms another interesting algorithm on leader election in distributed systems with crash failures by Stoller [5] popped up. It was judged to be a good, if not better, alternative to Singh's algorithm, but that had then already been implemented.

Much later, when an error was detected in the first implementation, and when we failed to repair this implementation based on Singh's article, we decided to try an implementation based on Stoller's algorithm. Here it is important to notice that the failure of Singh's algorithm lies solely in the problem of adapting the algorithm to the Erlang environment, not in the algorithm itself. As an example, Singh's algorithm only deal with one election round, in the gen_leader a new election should be initiated when the elected leader fail. It is often the case that algorithms described in articles have assumptions and preconditions that are not fulfilled by the target system, such as communication behavior and specific network topologies. It is also often the case that the target system requires additional functionality that is not included in the algorithm, such as interface functions and error handling. Therefore, changes to the algorithm are necessary. When dealing with complex algorithms, such changes are dangerous, since one easily introduces an error, which was exactly what had happened in the gen_leader

case.

Stoller's algorithm, is based on a pre-known set of participating processes with a globally known priority order. The algorithm also depends on the fact that there exist a mechanism for detecting *inactive* processes, for this we can use the Erlang *Monitor*. The basic algorithm is both simple and elegant. When a process is started, it first checks whether a process with higher priority is active. If such a process exists, the process simply waits for one of those processes to become the leader. If, on the other hand, the present process is the active process with highest priority, the process itself tries to become the leader. Becoming the leader is done by making sure that all processes with lower priority either are aware of its existence or are inactive. When all processes with lower priority are informed, the process announces itself as the leader. Periodically, the elected leader polls the inactive processes, if one of the inactive processes is activated, the election process is simply restarted.

There are actually two different algorithms described in Stoller's article, one with synchronous message passing and one with asynchronous message passing. What is perhaps a bit surprising, and at the same time shows how difficult it is to select a good candidate algorithm for implementation, is that we choose the synchronous algorithm, even though Erlang has asynchronous communication. A more careful reading of the article reveals however, that the difference between the synchronous and the asynchronous algorithm lies mostly in how the failure detection works (how node failures are detected and reported). The Erlang monitor works in the same way as the failure detection with synchronous message passing. This shows that it is important to have a thorough understanding of the inner workings of the implementation language.

We illustrate in detail how the algorithm works by an example with three participation processes in Fig. 1. The processes are named A,B and C, with priority $A > B > C$, i.e. A has highest priority.

Unfortunately, this algorithm does not behave as is required by a leader election in this case. The requirements for the leader election implementation is that (1) it should quickly elect a leader among the active participating processes, (2) the elected process stays the leader until it fails and (3) when the leader fails, a new process should be elected automatically. The algorithm presented by Stoller fulfills (1) and (3), but fails on (2). Instead whenever an inactive process is activated, a new round of elections is started, electing the process with highest priority as the leader. This is both time consuming and inefficient from a message complexity point of view, so in order to use this algorithm we have to change its behavior.

```
1. A,B and C are all activated at the same time
    - C: starts monitoring A and B,
      B: starts monitoring A.
    - A: no higher prioritized process alive,
         starts monitoring B,
         sends a 'halt'-message to B
    - B: receives a 'halt'-message,
         replies with an 'ack'-message
    - A: receive 'ack' from B,
         starts monitoring C,
         sends a 'halt'-message to C
    - C: receives a 'halt'-message,
         replies with an 'ack'-message
    - A: receive 'ack' from C,
         all processes notified so A is the leader,
         sends 'ldr'-message to B and C
    - B,C: receive 'ldr'-message from A,
           accepts A as the leader.

2. A and B are active and A is the elected leader,
   C is activated.
    - A: periodically sends a 'norm'-message to C
    - C: receives a 'norm'-message from A,
         replies with a 'notnorm'-message
    - A: receives a 'notnorm'-message,
         restarts the election procedure,
         no higher prioritized process alive,
         starts monitoring B,
         sends a 'halt'-message to B
         ... (as in situation 1)

3. A and C are inactive, B is active.
    - B: starts monitoring A
    - B: receives a 'DOWN,A'-message from monitor,
         no higher prioritized process alive,
         starts monitoring C,
         sends 'halt'-message to C
    - B: receives a 'DOWN,C'-message from monitor,
         all processes notified so B is the leader,
         sends 'ldr'-message to A and C

4. B and C are active and B is the elected leader,
   A is activated.
    - A: no higher prioritized process alive,
         starts monitoring B,
         sends a 'halt'-message to B
         ... (as in situation 1)
```

Figure 1: Examples – Original behavior

We made this change in two steps, first we changed the algorithm such that no new election would be started if a process with lower priority than the leader was activated. This change is fairly straightforward, and just requires a small modification to the behavior when a newly activated process is polled by the elected leader. Instead of restarting the election process, the newly activated process is informed of who the leader is. If we reconsider the examples in Fig. 1 situations 1, 3 and 4 are not changed, but in situation 2 we avoid a re-election and instead proceed as in Fig. 2.

```
2. A and B are active and A is the elected leader,
   C is activated.
    - A: periodically sends a 'norm'-message to C
    - C: receives a 'norm'-message from A,
         starts monitoring A,
         replies with a 'notnorm'-message
    - A: receives a 'notnorm'-message,
         sends a 'ldr'-message to C
    - C: receives a 'ldr'-message from A,
         accepts A as the leader
```

Figure 2: Examples – Situation 2 without re-election

In addition we wanted to do something similar when a node with higher priority than the present leader is activated. This however turned out to be much more complicated. The reason for the complexity is the fact that a node with high priority is likely to conclude that there are no processes active with a higher priority and therefore initiates a new election. (Note however that this behavior is required, otherwise an election would never be initiated in the first place.) The basic trick here is to make sure that a process that knows who the leader is will not surrender to the newly activated process, instead it sends a reply saying who (he thinks) is the leader. In this way, also a newly activated process with high priority can be informed of who is the leader. The newly activated process finally confirms the leadership with the leader. Nevertheless, there are still many things that can go wrong, especially in situations where the present leader fails in the middle of the information phase. If we yet again reconsider the examples in Fig. 2, we see that situations 1 and 3 work as before, but as expected we do not get a re-election in situation 4. This can be seen in Fig. 3

We also made some changes that did not affect the observable functionality, but which reduced the number of messages sent by the system.

3 Implementation and Testing

We first implemented the algorithm as a gen_server behavior, in order to quickly evaluate if it was working as intended. Having corrected several minor errors, most of them related to messages that were not treated in all situations, we felt fairly sure that this algorithm would work inside the gen_leader. Replacing the

```
4. B and C are active and B is the elected leader,
   A is activated.
    - A: no higher prioritized process alive,
         starts monitoring B,
         sends a 'halt'-message to B
    - B: receives a 'halt'-message,
         replies with an 'hasLeader,B'-message
    - A: receive 'hasLeader,B' from B,
         starts monitoring B,
         sends an 'isLeader'-message to B
    - B: receive 'isLeader' from A,
         sends 'ldr'-message to A
    - A: receive 'ldr'-message from B,
         accepts B as the leader.
```

Figure 3: Examples – Situation 4 without re-election

old algorithm was relatively easy, the only problem was the separation into a *safe_loop* (where the process execute during elections) and a working *loop* (where the process execute when a leader is elected and which basically is the same as the loop in a generic server). This separation made it possible to do some simplifications in the message receiving code, and introduced a couple of new errors.

Another problem is the fact that the new algorithm is fundamentally different from the old one. This leads to some problems when trying to be compatible with the existing implementation. In particular we realized that the *query*-functions alive (which returns all active participating processes) and down (which returns all inactive processes) could not be implemented. This is because the new algorithm does not keep track of this information at all times, so the information returned by these functions is not reliable. Except from this, we managed to implement the algorithm without changes to the interface.

Leader election is a well-known and clearly defined problem, which means that the requirements are also well defined: (1) Eventually, a leader should be elected, and (2) At most one of the participants is considered the leader. These properties are also stated in Stoller's article [5]. We tested the implementation with two different methodologies, first we used the method with abstracted traces, as we describe in [1] and second we used Erlang QuickCheck presented in [3].

3.1 Testing with trace recording

The built-in trace functionality in Erlang is a very useful tool when testing an implementation. However, the raw trace data has a tendency to get very verbose, containing lots of events and also a lot of data per event. Manual inspection of traces is therefore often both tedious and time consuming, and alternative approaches have been proposed. In [2], one approach is presented where *abstraction functions* are applied to state based trace data, in order to remove unnecessary data and reduce the state space. The state space is reduced since different con-

crete states will be reduced to the same abstract state when the abstraction function is applied. While collapsing different concrete states to the same abstract state, cyclic behaviors can be detected. The abstract state space is also visualized, something that gives a good intuition about the inner workings of an implementation.

This *abstract trace* approach is taken even further in [1], where we demonstrate the effectiveness of the method by testing the first leader election implementation [6] based on Singh's algorithm. In [1] we also introduce a small language for constructing abstraction functions, as well as checking LTL-properties for the abstract state space. To test the leader election implementation we stimulated the system by arbitrarily killing and reviving nodes, and by arbitrarily delaying messages sent between processes.

This test method initially revealed a couple of trivial implementation errors, but when those were corrected all tests where executed without errors. That is the new implementation passed all the tests, the same tests during which the previous implementation failed in two cases. However, this test method does not change the scheduler in the runtime system, and since the Erlang scheduler is deterministic, it seemed quite possible that there exist execution paths not exercised by the trace recording testing technique.

3.2 Testing with QuickCheck

Therefore we decided to also test the implementation with Erlang QuickCheck, presented by Arts and Hughes in [3]. QuickCheck is a property-based tool for random testing. Developers write properties in a restricted logic, and then invoke QuickCheck to test the property in a large number of cases. QuickCheck tests concurrent programs by collecting a trace of events, which should have the properties the developer specifies. The events are defined by instrumenting the code under test with calls to the QuickCheck function `event`. QuickCheck delays these calls randomly, thus in effect overrides the Erlang scheduler and forces a random schedule on the system under test. This can elicit faulty behavior that would appear only very rarely with the normal scheduler, which is exactly what we want to test here. Testing the leader election implementation was done by randomly killing and reviving leader election processes.

Using QuickCheck to test the second implementation, we could not produce any trace where the properties were violated. Nevertheless, and much to our surprise, we could observe some faulty behavior, namely that a leader election process crashed unexpectedly from time to time. This did not lead to any faulty behaviour, but it indicated that something was wrong.

Closer analysis revealed a very tricky error, which would have been extremely unlikely to be found without control of the scheduling. The problematic situation occurs whenever a process A is about to contact another process B. To do this in a controlled way, process A first request a monitor on process B before sending the message. What can occur now is that process B is down when process A requests the monitor, but alive just some time later when process A send the message. In this case, process A receive both a failure-notification and a message reply. This

	Killed nodes	**in election**	**as leader**	**surrendered**
QuickCheck	1601	379	102	1120
QC average	19.3	4.6	1.2	13.4
Trace rec.	101	4	11	86

Table 1: Coverage results

situation was overseen in the implementation and lead to a crash. Luckily, the error could easily be corrected.

In this example we can see how important it is to have control of the scheduling, since this situation occurred frequently (like once every 150 tests) while testing with QuickCheck, but could not at all be observed when we tested the implementation with the trace recording technique.

3.3 Coverage

When working with test methods, the issue of coverage is central. Coverage should provide a measure of how exhaustively one has exercised the system, and is therefore important when evaluating the results of testing. Though it is very rare that a coverage measure can tell when we have tested enough, rather the coverage measure will warn of potential situations when we have *not* tested enough.

In [1] we discuss some coverage measures for the trace recording technique, but those measures mostly deal with quantities in the abstracted state space and are hard to compare with the QuickCheck tests. Instead we choose to look at how many nodes that where killed, and at what stage in the election process the node were killed.

In Tab. 1 we can see: the coverage result (labeled QuickCheck) for a QuickCheck run with 5 nodes, average numbers (labeled QC average), and as a comparison results for a run with the trace recording technique (labeled Trace rec). The first column shows the total number of killed nodes, second column the number of nodes killed during an election, third column the number of nodes killed when elected as leader, and fourth column the number of nodes killed when being surrendered to a leader. In the coverage results we can note a difference between the two techniques, since we do not influence the scheduler in the trace recording technique it is quite rare that we manage to kill a node in the middle of the election process (merely 4% of the kills) compared to the QuickCheck approach where this happens a lot more frequently (almost 25 % of the kills).

Other coverage measures that are often discussed include *code coverage* and *path coverage*. Code coverage is a very basic coverage measure, that only measures whether (or how many times) a certain line of code has been executed. This simple measure is useless here, since it is the complicated interaction of several

different instances of the implementation that is studied. Path coverage is therefore more interesting, since it measure how many different paths that has been taken through the code. Unfortunately however, path coverage is hard to define in a functional language such as Erlang since paths does not exist in the same way as for an imperative language like C or Java.

4 Discussion

Implementing a new leader election algorithm was very interesting from more than one point of view. Not only is it a challenging intellectual problem, it also highlights several interesting and problematic situations that may occur in industry. For the majority of algorithmic problems that arise in practical software development today, there exist books and papers describing possible solutions. For a software engineer, it is often a non-trivial task to first find the right source of information, and then adopt the described solution to the specific setting at hand. Often software errors are made because (1) the wrong algorithms were chosen, or (2) the right algorithms were adapted in the wrong way.

Why is it such a hard problem to choose a good algorithm? Algorithm descriptions, and then especially formally verified algorithms, are often presented in a theoretical way and work only in a specific setting. It is often the case that the prerequisites stated in the article do not fit into the implementation language. It is also often the case that changes must be made to the algorithm in order to fulfill the specific requirements, such changes include error-handling and interface. Therefore it is a hard but also crucial problem to select a good algorithm. It is a task that require not only a thorough understanding of the problem, but also a good insight in the inner workings of the implementation language.

One example of this is the error found with QuickCheck, our erroneous implementation closely followed the algorithm in the paper. Nevertheless, the implementation turned out to be incorrect. Does this mean that the same error is also present in the article? No, Stoller's article [5] is not very precise about the semantic assumptions made regarding link requests between processes. Therefore, one has to assume that there is a difference in how the monitoring works, and that this is the source of the error. This clearly shows the difficulties of bridging the semantics from the article, where underlying assumptions often hide important and problematic issues, to the implementation language.

Verifying fault-tolerant distributed systems is an extremely difficult task. It is difficult and time consuming to use verification techniques such as model checking, instead testing is the commonly used method. Here, we have used two different testing techniques. In many ways these techniques are rather similar; both use random testing, and both methods use traces. The big difference between the methods are the way we control the scheduler, which in turn affects the execution paths explored in the tests. The concrete test results show that both methods are useful, we found an error with QuickCheck that was not found with the trace recording technique. On the other hand when writing the implementation it was very useful to see the visualizations from the trace recording technique, both to correct errors and to gain insight in the implementation.

Our work resulted in a new implementation of the generic leader behavior. This implementation is thoroughly tested and no errors could be identified. For some very critical applications, one might want to invest in a formal verification of the presented application, but most applications would not require such thorough mathematical analysis.

5 Acknowledgments

Thanks to Ulf Wiger, co-author of the first gen_leader implementation, and John Hughes, implementer of Erlang QuickCheck. Also thanks to Koen Claessen for providing valuable comments and ideas.

References

[1] T. Arts, K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. In *Lecture Notes in Computer Science*, volume 3395, pages 140 – 154. Springer, Feb 2005.

[2] T. Arts and L.-Å. Fredlund. Trace analysis of Erlang programs. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 16–23. ACM Press, 2002.

[3] T. Arts and J. Hughes. Erlang/QuickCheck. In *Ninth International Erlang/OTP User Conference*, Nov. 2003.

[4] G. Singh. Leader election in the presence of link failures. In *IEEE Transactions on Parallel and Distributed Systems, Vol 7*. IEEE computer society, 1996.

[5] S. Stoller. Leader election in distributed systems with crash failures. Technical Report 481, Computer Science Dept., Indiana University, May 1997. Revised July 1997.

[6] U. Wiger. Fault tolerant leader election. http://www.erlang.org/.

Paper 3

A More Accurate Semantics for Distributed Erlang

This paper was written together with Lars-Åke Fredlund for the Sixth ACM SIGPLAN Erlang Workshop, in Freiburg, Germany, October 2007. The paper included here has a few minor corrections and is also typeset in a different style.

A More Accurate Semantics for Distributed Erlang*

Hans Svensson[1], and Lars-Åke Fredlund†[2]

[1] Chalmers University of Technology, Göteborg, Sweden
hanssv@cs.chalmers.se

[2] Facultad de Informática, Universidad Politécnica de Madrid, Spain
fred@babel.ls.fi.upm.es

Abstract

In order to formally reason about distributed Erlang systems, it is necessary to have a formal semantics. In a previous paper we have proposed such a semantics for distributed Erlang. However, recent work with a model checker for Erlang revealed that the previous attempt was not good enough. In this paper we present a more accurate semantics for distributed Erlang. The more accurate semantics includes several modifications and additions to the semantics for distributed Erlang proposed by Claessen and Svensson in 2005, which in turn is an extension to Fredlund's formal single-node semantics for Erlang. The most distinct addition to the previous semantics is the possibility to correctly model disconnected nodes.

Categories and Subject Descriptors: D.3.1 [*Formal Definitions and Theory*]

General Terms: Languages, Theory, Verification

Keywords: Erlang, semantics, distributed systems, verification, model checking

1 Introduction

Software systems written in Erlang are often running in a distributed environment, and are often highly concurrent and dynamic in nature. Something that has lately become even more emphasised by the introduction of multi-core and SMP‡ computers. And although Erlang with its *Concurrency Oriented Programming* paradigm is particularly suited for writing such applications, experience still

†The author was supported by a Ramón y Cajal grant from the Spanish Ministerio de Educación y Ciencia, and the DESAFIOS (TIN2006-15660-C02-02) and PROMESAS (S-0505/TIC/0407) projects.

‡Symmetric Multiprocessing

shows that concurrent and fault-tolerant software is hard to write, test and verify. Because of this, several approaches have been proposed for testing [6, 7, 9, 21, 5] Erlang programs and also a lot of work has been done on formal verification of Erlang programs [19, 18, 3, 16].

Since the history of Erlang starts in industry, and not in a university, Erlang is very much defined in terms of its implementation. However, when working with formal verification a formal semantics is almost indispensable, and doing verification without a formal semantics is really hard. In 1999 Fredlund proposed a formal semantics for Erlang [14], and the semantics is described in detail in [15]. Fredlund's semantics is a small-step operational semantics. It is constructed in a simple, easy to understand layered fashion. The semantics has been used as a basis in several different verification projects, such as semi-formal verification of Erlang code [17] and model checking a resource manager [18]. Fredlund's semantics has also been a basis for the development of a theorem prover [18] and a translation of Erlang into a language that can be model checked [4].

In 2004 two previously unknown errors was discovered in an open source implementation in Erlang of a leader election algorithm [22, 5]. It turned out that both errors were caused by difficult to foresee chains of events related to the arrival order of messages in the distributed environment. It also turned out that the errors were specific to a *multi-node* setting. That is, the errors could only be found when different parts of the system run on different nodes. Thus, contrary to the Erlang idea that distribution should be transparent, there is a real behavioral difference between single-node and multi-node systems.

During the analysis of the errors in the leader election implementation, we realized that it is impossible to reason about this type of errors in Fredlund's semantics. The semantics does not contain the concept of nodes, and all processes are localized at the same run-time system. This also means that it is impossible to detect this kind of multi-node errors in a model checker based on Fredlund's single-node semantics.

Therefore we proposed an extension of the semantics into a distributed (multi-node) semantics for Erlang [11]. In that paper we added a distributed layer on top of the single node semantics, and were able to successfully model multi-node programs in the extended semantics. In the paper we also warned the community of potential pit-falls with testing and verification using a single-node semantics.

The semantics for distributed Erlang have since been used in the implementation of a model checker for Erlang (McErlang, [16]). From the work with the model checker we could observe that the proposed multi-node semantics was incomplete. (With incomplete we mean that there are possible behaviors in the Erlang run-time system, which can not be described by the semantics.) The incompleteness stems mainly from the Erlang behavior in the case of *node disconnect*. Two Erlang nodes can become disconnected from each other if the link between them fails, when this happens both involved nodes regard the other node as dead. This does have some interesting consequences when the nodes later re-connect. Such behavior can not be modeled in the multi-node semantics we proposed earlier. Since the publication of the semantics for distributed Erlang we have also discovered a few minor errors in the semantics as well as some rather

embarrassing inelegancies in the presentation of the semantic rules. Therefore we have re-structured and extended the distributed layer and now propose a more accurate multi-node semantics.

In order to make this paper self contained and the presentation as easy as possible to follow, some material from the previous paper by Claessen and Svensson ([11]) is presented here as well. This especially concerns Sect. 2 with the single-node semantics, which naturally has not changed, and Sect. 3 where the motivation for the multi-node semantics is still at least partly the same.

Contributions The contribution of the paper is a clear and self contained presentation of a more accurate distributed (multi-node) semantics for Erlang. In addition to previous attempts to present a semantics for distributed Erlang, this presentation includes together with some other modifications and additions novel semantic rules to correctly express node disconnects. We also present and informally argue for desirable properties of the proposed semantics. The multi-node semantics has already proved to be useful, when used as the basis for a model checker for Erlang (McErlang).

Summary Sect. 2 contains an introduction to Fredlund's single-node semantics. In Sect. 3 we present some motivating examples, as well as a description of situations where Fredlund's Erlang semantics lacks expressive power. In Sect. 4 we provide an extension to Fredlund's semantics, where we add another layer on top of the existing single-node semantics in order to introduce the full distributed behavior. In Sect. 5 we specify desirable properties of the multi-node semantics, and argue why they are fulfilled by the presented semantics. Some of the design decisions in the extended semantics are further discussed in Sect. 6, related approaches to the problem are described in Sect. 7 and finally, we conclude in Sect. 8.

2 Original semantics

In [15], Fredlund gives a complete presentation of a small-step operational semantics for Erlang. Here we will highlight some of the most important aspects, with enough details to be able to understand the presentation of the extended semantics. Fredlund's single-node semantics is presented for a subset of Erlang, that is in short standard Erlang without: modules, nodes, floats, references, binaries, ports and the catch-expression. Some of the process' internal state has also been omitted: there are no process dictionaries, no group leader or processes groups and name-registration for processes is also not included.

All definitions and rules presented in this section are taken from Fredlund's presentation of the semantics [15], with the exception that we in a few cases leave out details not relevant for this article in order to make the presentation clearer. Fredlund's semantics is separated into two parts; one functional part, with evaluation of expressions and one concurrent part where processes are spawned and messages are sent and delivered. Fredlund's single-node semantics is presented here in roughly the same order as in the original presentation [15], starting with *expression evaluation rules* then defining processes and finally stating *process evaluation rules*.

$$send_0\ \frac{}{pid!v \xrightarrow{pid!v} v}$$

$$send_1\ \frac{e_1 \xrightarrow{\alpha} e_1'}{e_1 ! e_2 \xrightarrow{\alpha} e_1' ! e_2}$$

$$send_2\ \frac{e \xrightarrow{\alpha} e'}{v ! e \xrightarrow{\alpha} v ! e'}$$

$$receive\ \frac{\begin{array}{c}\forall i.\neg(\mathsf{qmatches}\ q\ m_i) \\ \exists i.((\mathsf{result}\ v\ m_i\ e') \wedge \forall j.j < i \Rightarrow \neg(\mathsf{matches}\ v\ m_j))\end{array}}{\texttt{receive}\ m\ \texttt{end} \xrightarrow{\mathsf{read}(q,v)} e'}$$

Figure 1: Expression evaluation rules

Definition 1 Erlang expressions are ranged over by $e \in \mathsf{erlangExpr}$; Erlang values (non-reducible expressions) are ranged over by $v \in \mathsf{erlangVal}$.

The semantics is provided in terms of transition rules on the format

$$\frac{t_1 \xrightarrow{\alpha_1} t_1'\ \ldots\ t_n \xrightarrow{\alpha_n} t_n' \qquad \varphi_1\ \ldots\ \varphi_m}{t \xrightarrow{\alpha} t'}$$

where each φ_i is a logic side-condition that does not refer to any transition relation.

Definition 2 The expression actions, ranged over by $\alpha \in \mathsf{erlangExprAction}$, are:

$\gamma ::=$	τ		computation step
$\mid$	$pid\,!\,v$		output
$\mid$	$\mathsf{exiting}(v)$		exception
$\mid$	$\mathsf{read}(q, v)$		read from queue
$\mid$	$\ldots$		

Definition 3 The expression transition relation $\rightarrow$: $\mathsf{erlangExpr} \times \mathsf{erlangExprAction} \times \mathsf{erlangExpr}$, written $e_1 \xrightarrow{\alpha} e_2$ when $\langle e_1, \alpha, e_2\rangle \in \rightarrow$, is the least relation satisfying the transition rules in [15].

In Fig. 1 we have listed Fredlund's rules for evaluation of send and receive at the expression level. The send-rules are fairly straightforward, both terms are evaluated until finally a $pid!v$-action is generated. The receive-rule is more complicated, and won't be explained in detail. The intuition is that q is a prefix to the complete message queue, and none of the messages in that prefix matches any of the patterns in m. Also, there exist a pattern in m, such that it is the first one

to match v, and when substituting v according to that pattern its corresponding expression become e'.

Next we need to formalize the notion of processes, which encapsulate Erlang expressions, and the notion of Erlang systems, which are collections of processes. Erlang processes, ranged over by $p \in \mathsf{erlangProcess}$, are either live or dead. The dead processes are introduced to make it easier to reason about the semantics of linked processes. Processes that are dead still perform some actions; they will eventually inform linked process about their termination, and they do respond to received link signals.

Definition 4 An Erlang mailbox is a queue data structure, in theory unbound, thus it can store any number of messages. Mailboxes are ranged over by $q \in \mathsf{erlangQueue}$, and ϵ denotes the empty queue.

Definition 5 A **live** *Erlang process* ($\mathsf{erlangLiveProcess} \subset \mathsf{erlangProcess}$), is a quintuple: $\mathsf{erlangExpr} \times \mathsf{erlangPid} \times \mathsf{erlangQueue} \times \mathcal{P}(\mathsf{erlangPid}) \times \mathsf{erlangBool}$, written $\langle e,\ pid,\ q,\ pl,\ b \rangle$ such that

- e is an Erlang expression,
- *pid* is the process identifier of the process,
- q is a message queue,
- *pl* is a set of process identifiers (a set of links with other processes),
- b is a boolean determining how process exit notifications are handled.

Definition 6 A terminated (dead) Erlang process ($\mathsf{erlangDeadProcess} \subset \mathsf{erlangProcess}$) is a tuple: $\mathsf{erlangPid} \times \mathcal{P}\ (\mathsf{erlangPid} \times \mathsf{erlangVal})$, written $\langle pid,\ plm \rangle$, where

- *pid* is the process identifier of the process,
- *plm* is a set of tuples, combining process identifiers with a notification value that should be sent to the corresponding process.

Definition 7 An *Erlang system*, ranged over by s, is either a singleton process or a combination of systems s_1 and s_2, written as $s_1 \parallel s_2$.

Intuitively, the composition of processes into Erlang systems could be thought of as a set of processes. The $\parallel$ operator is commutative and associative. When there is no risk for confusion, we omit the linked processes parameter and the boolean flag from the live processes, that is they are written as $\langle e, pid, q \rangle$. The *signals* are items of information transmitted between a sending and a receiving process. A *system action*, committed by an Erlang system is either a silent action, an input action or an output action. We should also define the system transition relation.

$$\mathit{silent}\ \frac{e \xrightarrow{\tau} e'}{\langle e, pid, q, pl, b\rangle \xrightarrow{\tau} \langle e', pid, q, pl, b\rangle}$$

$$\mathit{output}_1\ \frac{e \xrightarrow{pid'\,!\,v} e' \qquad pid' \neq pid}{\langle e, pid, q, pl, b\rangle \xrightarrow{pid'\,!\,\mathsf{message}(v)} \langle e', pid, q, pl, b\rangle}$$

$$\mathit{output}_2\ \frac{e \xrightarrow{pid\,!\,v} e'}{\langle e, pid, q, pl, b\rangle \xrightarrow{\tau} \langle e, pid, q \cdot v, pl, b\rangle}$$

$$\mathit{input}\ \frac{}{\langle e, pid, q, pl, b\rangle \xrightarrow{pid\,?\,\mathsf{message}(v)} \langle e, pid, q \cdot v, pl, b\rangle}$$

$$\mathit{link}\ \frac{}{\langle e, pid, q, pl, b\rangle \xrightarrow{pid\,?\,\mathsf{link}(pid')} \langle e, pid, q, pl \cup \{pid'\}, b\rangle}$$

$$\mathit{term}\ \frac{}{\langle e, pid, q, pl, b\rangle \xrightarrow{\tau} \langle pid, \{\langle P, \mathsf{normal}\rangle | P \in pl\}\rangle}$$

Figure 2: Process evaluation rules

Definition 8 The signals, ranged over by $sig \in \mathsf{erlangSignal}$ are:

$sig ::=$	$\mathsf{message}(v)$	message
$\mid$	$\mathsf{link}(pid)$	linking with process
$\mid$	$\mathsf{unlink}(pid)$	unlinking process
$\mid$	$\ldots$	

Definition 9 The system actions, ranged over by $\alpha \in \mathsf{erlangSysAction}$ are:

$\alpha ::=$	τ	silent action
$\mid$	$pid\,!\,sig$	output action
$\mid$	$pid\,?\,sig$	input action

Definition 10 The system transition relation $\rightarrow$: $\mathsf{erlangSystem} \times \mathsf{erlangSysAction} \times \mathsf{erlangSystem}$, written $s_1 \xrightarrow{\alpha} s_2$, is the least relation satisfying the transition rules in [15]. Some of those rules are listed here in Fig. 2 and Fig. 3.

The rules in Fig. 2 show how processes perform a computation step, terminates and sends and receives messages. Note that messages sent to the same process are delivered immediately (output_2). Also note that messages to other processes are transferred to the above layer by a visible ($pid'!\mathsf{message}(v)$) system action. The rules in Fig. 3 show how processes communicate and how computations are interleaved, note that the communication rules also exist in a symmetric version

$$com\ \frac{s_1 \xrightarrow{pid\ !\ sig} s_1' \qquad s_2 \xrightarrow{pid\ ?\ sig} s_2'}{s_1 \parallel s_2 \xrightarrow{\tau} s_1' \parallel s_2'}$$

$$interleave\ \frac{s_1 \xrightarrow{\tau} s_1' \qquad \mathsf{pids}(s_1') \cap \mathsf{pids}(s_2) = \emptyset}{s_1 \parallel s_2 \xrightarrow{\tau} s_1' \parallel s_2}$$

Figure 3: Process communication rules

where the roles of s_1 and s_2 are interchanged. The function pids used in the *interleave*-rule, simply returns all process identifiers in the Erlang system. This concludes the introduction to the original semantics, an example with Fredlund's single-node semantics in use is presented in Sect. 3.

3 Motivation

The motivation for creating a multi-node semantics for Erlang comes from observations made during research projects. There were some cases when we did not understand the behavior of our Erlang programs and other cases when we were just curious about how the run-time system is implemented. When we had figured out how things actually worked, we realised that the existing single-node semantics was not expressive enough to describe the problematic situations. Below we describe two motivating examples, where we have quite ordinary situations in which the single-node semantics is not expressive enough.

3.1 Message reordering

```
procA() ->
    PidC = spawn(?NODE2,?MODULE,procC,[]),
    PidB = spawn(?NODE1,?MODULE,procB,[PidC]),
    PidC ! hello,
    PidB ! world.

procB(PidC) ->
    receive X -> PidC ! X end.

procC() ->
    receive X -> ok end.
```

Figure 4: Erlang program - Message reordering

In our work with a leader election protocol [5], we saw several cases where problems arise due to unforseen order of events. Especially problematic were

situations when messages arrived in what was thought to be an impossible order. To investigate this problem further, we constructed the Erlang program listed in Fig. 4. This Erlang program (process A) first spawns two processes (C and B, and passes the process identifier of C to B) and then sends a message, *hello*, directly to process C. Next the program sends another message, *world*, to process B. When process B receives a message, it is immediately re-sent to process C. Process C does only one thing, namely receives one message. Intuitively, process C will receive the message *hello*, since it is sent directly from A to C. However, in the fundamental ideas behind Erlang [1] the only thing said about message order is *'Message passing between a pair of processes is assumed to be ordered'*. This means that without violating this property *world* should be able to arrive before *hello*, since we have no guarantees for the relative message order when the messages are sent on different routes. This understanding of possible message orderings is further confirmed in the natural language semantics for Erlang (draft) by Barklund and Virding [8] (Sect. 10.5.4): *'It is assured (through the rules of signals, cf. §10.6.2) that if a process P_1 dispatches two messages M_1 and M_2 to the same process P_2, in that order, then message M_1 will never arrive after M_2 at the message queue of P_2. Note that this does not guarantee anything about in which order messages arrive when a process sends messages to two different processes...'*

The possible executions are depicted in Fig. 5.

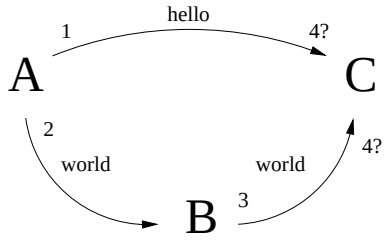


Figure 5: Message passing

The program in Fig. 4 was executed in three different situations

1. A,B and C where executed in the same run-time system.
2. A,B and C where executed on the same physical machine, but in separate run-time systems.
3. A,B and C where executed on three different physical machines connected via a 100 MBit Ethernet network, thus running in separate run-time systems.

The results were somewhat surprising. If the execution would follow the intuition, *hello* should always arrive first; if the Erlang ideas where implemented faithfully we should see both *hello* and *world* arriving first in all three situations. However, in situation (1) *hello* always arrive first and in situations (2) and (3) we could see both *hello* and *world* arriving first. The conclusion is that the Erlang run-time

1. Initial system:

$P_0 = \langle \texttt{PidC} = \texttt{spawn}(\texttt{procC}, []) \ \ldots, p_0, \epsilon \rangle$

2. The only scheduler option is to spawn `procC`. After that, the only option is to spawn `procB` since the `receive` in `procC` (P_2) blocks. This results in three processes:

$P_0 = \langle \texttt{PidC} \,!\, \texttt{hello} \ldots, p_0, \epsilon \rangle$
$P_1 = \langle \texttt{receive}\ X \rightarrow \texttt{PidC} \,!\, X\ \texttt{end}, p_1, \epsilon \rangle$
$P_2 = \langle \texttt{receive}\ X \rightarrow \texttt{ok end}, p_2, \epsilon \rangle$

3. Only P_0 can make progress, since P_1 and P_2 are blocked on a `recieve` statement:

$P_0 = \langle \texttt{PidB} \,!\, \texttt{world}, p_0, \epsilon \rangle$
$P_1 = \langle \texttt{receive}\ X \rightarrow \texttt{PidC} \,!\, X\ \texttt{end}, p_1, \epsilon \rangle$
$P_2 = \langle \texttt{receive}\ X \rightarrow \texttt{ok end}, p_2, \epsilon \cdot \texttt{hello} \rangle$

4. Here we see that there is no way that `procC` can receive `world` before `hello`. This is because we have a 'match all' pattern (the single unbound variable `X`) in `procC` and any later arriving message is put last in the mailbox. Therefore, the next application of the receive-rule (Fig. 1) must read `hello` from the mailbox.

Figure 6: Hello World - Single-Node Execution

system implementation behaves differently in a local setting as compared to in a distributed setting (and also differently from the Erlang specifications, this is further disscussed in Sect. 6). This partly explain why errors such as those found in the leader election implementation [5] appear to be common.

Another reason is that Erlang programmers often think of their system in an event-based way: "First this process dies, then that process sends a message, then that message is sent...". In other words they have a conceptual model of the many possible orders in which the events can be generated. The semantics adds additional possibilities in the form of the possible orders in which the events actually arrive. This extra complexity may be hard to deal with and the speed with which messages are delivered allows programmers to often only think in terms of generated events. Thus, if one does not think carefully enough, it is easy to be misled and overlook something.

Message reordering in Fredlund's semantics

What happens if we try to analyze the program in Fig. 4 with Fredlund's single-node semantics? Since the single-node semantics does not include nodes, it is not too surprising that the program will behave as in situation (1) above, as we can see in Fig. 6. The desire to be able to describe also the behavior in situations (2) and (3) serves as the motivation for extending Fredlund's single-node semantics to be able to fully reason about distributed Erlang systems. This is especially important in case we use the semantics to produce a model, if certain situations are not present in the model, errors may be overlooked, and thus giving false confidence.

3.2 Disconnected nodes

Another interesting and potentially dangerous behavior of the Erlang run-time system occurs when *nodes disconnect* from each other. In the simple situation two nodes (it could be generalized to many nodes) become disconnected because one of them dies. This of course means that all processes on the failing node dies, and processes on the surviving node will be notified of this via the link-mechanism. This situation is not dangerous, and it could be simulated in the single-node semantics by grouping processes together at a meta level and then kill off a whole group of processes.

The potentially dangerous situation is when two processes become disconnected because the link (in the ordinary case, the network connection) between them breaks down. In that case both of the nodes continue to execute, and both nodes consider the other node to have failed(!). Thus the processes are informed of the failure of processes on the other node via the link-mechanism. Things then become really interesting when the nodes *re-connect*, because this happens without any notice to the running processes. This should be considered harmful since messages can be dropped silently (if the link mechanism is not used), which breaks the common assumptions about TCP/IP like communication in Erlang.

From a programmer's point of view this requires some extra caution, and as long as this behavior is taken into consideration it should not cause too much trouble. However if one is careless, and for example has a system with two processes running on different nodes (A and B), where A sends a stream of messages to B and only occasionally gets a reply from B. Then if neither A or B uses the link mechanism it could be the case that A sends a lot of messages that B never receives because the nodes are disconnected, and later the nodes are re-connected before A expects an answer. This behavior obviously can not be described in the single-node semantics.

It should be noted that this phenomenon is also acknowledged in Barklund and Virdings natural language semantics for Erlang [8] (Sect. 10.6.2): *'It is guaranteed that if a process P_1 dispatches two signals s_1 and s_2 to the same process P_2, in that order, then signal s_1 will never arrive after s_2 at P_2. It is ensured that whenever possible, a signal dispatched to a process should eventually arrive at it. There are situations when it is not reasonable to require that all signals arrive at their destination, in particular when a signal is sent to a process on a different node and communication between the nodes is temporarily lost.'*. In this context a message is a signal. That is, there are no promises regarding safe delivery (except no reordering), especially during temporary communication failures.

4 Distributed (Multi-Node) Semantics

In this section Fredlund's single-node semantics is extended, by adding a new layer of semantic rules, to a distributed (multi-node) semantics. By adding another layer on top of the existing semantics we can deal with all aspects of distribution without making more than a few marginal changes to the single-node semantics. One important implication of this is that everything that is defined in terms

of the single-node semantics is still valid in the distributed semantics under the restriction that the system is local, i.e. running on the same node.

The distributed layer of the semantics is presented in three steps; Firstly, we add the possibility to *spawn* processes on other nodes. To be able to do this, we have to extend the concept of *Erlang systems* to *Erlang Run-Time systems*, i.e. a single node, and also *Erlang Multi-node systems* which are collections of nodes forming complete distributed systems. We also need to make some minor changes to the single-node semantics. Secondly, we need new rules for communication between processes on different nodes (i.e. different run-time systems). These communication rules should have the properties described in Sect. 3, and thus enable certain message reordering as well as introduce the possibility to drop messages in the case of node disconnect. Thirdly, we add the concept of nodes that die and get restarted. We also need to extend the linking mechanism in order for it to work also in the distributed semantics.

4.1 Nodes

Before we can define semantic rules for multi-node Erlang systems we have to introduce *node identifiers*. The node identifier could be any unique identifier. For the sake of simplicity, we can assume that they are integers. We also need two functions that returns node identifiers:

Definition 11 Let the function node(erlangPid) return the node identifier for a given process identifier and let node(erlangSystem) return the node identifier for an Erlang system.

4.2 Node message queues

The message ordering induced by a single-node semantics is too deterministic; certain message reorderings are not considered. We achieve the distributed ordering by introducing one message queue *per node*, holding all messages currently 'in transit' to that node.

Definition 12 An *Erlang node message queue*, ranged over by $nq \in$ erlangNodeQueue, consists of a finite sequence of triplets $v_x = (from_x, to_x, sig_x)$: $v_1 \cdot v_2 \cdot \ldots \cdot v_n$, where ϵ is the empty sequence, $(\cdot)$ is concatenation and $(\setminus)$ is deletion of the first matching triplet, e.g.

$$\begin{aligned} nq &= (a_2, b_1, c_1) \cdot (a_1, b_2, c_1) \cdot (a_1, b_2, c_2) \cdot (a_1, b_2, c_1) \setminus (a_1, b_2, c_1) \\ &= (a_2, b_1, c_1) \cdot (a_1, b_2, c_2) \cdot (a_1, b_2, c_1) \end{aligned}$$

4.3 Run-Time systems

Next, we define the concepts of live and dead run-time systems.

Definition 13 A **live** *Erlang Run-Time system (ERTS)*, ranged over by $r \in$ erlangRuntimeSystem is a triplet: erlangSystem $\times$ erlangNodeName $\times$ erlangNodeQueue, written [s, node, nq], where:

- s is the Erlang system at node *node*.
- *node* is the node identifier (name).
- nq is a node message queue.

Definition 14 A **dead** *Erlang Run-Time system* is a tuple: $\mathsf{erlangNodeName} \times \mathcal{P}(\mathsf{erlangPid})$, written $[\![node,\ npl]\!]$, where:

- *node* is the node identifier.
- *npl* is a set of process identifiers (consisting of all processes on the node *node*).

An example of a dead ERTS is: $[\![n_1,\ \{p_1, p_5, p_{13}\}]\!]$ where the node identifier is n_1, and the processes that has executed on n_1 are p_1, p_5 and p_{13}. Note also that $\mathsf{node}(p_5) = n_1$.

Definition 15 An *Erlang Multi-node system* (EMNS) is either a single ERTS or a composition of Erlang Multi-node systems n_1 and n_2, written as $n_1 \parallel n_2$.

Note that here we have chosen to use the same notation ($\|$) for composition of Erlang Multi-node systems as for the composition of Erlang systems in the original semantics. This is to illustrate that they are similar in behavior. Moreover, there is little risk for confusion.

4.4 Changes to the single-node semantics

$$com\ \frac{s_1 \xrightarrow{pid!_{from} sig} s_1' \quad s_2 \xrightarrow{pid?sig} s_2' \quad \mathsf{node}(pid) = \mathsf{node}(from)}{s_1 \parallel s_2 \xrightarrow{\tau} s_1' \parallel s_2'}$$

Figure 7: New com-rule

$$spawn_{local}\ \frac{e \xrightarrow{\mathsf{spawn}(n,f,[v_1,\ldots,v_m]) \rightsquigarrow \{result,pid'\}} e' \qquad pid \neq pid' \qquad \mathsf{node}(pid) = n}{\langle e, pid, q, pl, b\rangle \xrightarrow{\tau} \langle e', pid, q, pl, b\rangle \parallel \langle f(v_1,\ldots,v_m), pid', \epsilon, \emptyset, false\rangle}$$

$$spawn_{dist}\ \frac{e \xrightarrow{\mathsf{spawn}(n,f,[v_1,\ldots,v_m]) \rightsquigarrow \{result,pid'\}} e' \qquad \begin{array}{c} pid \neq pid' \\ \mathsf{node}(pid) \neq n \\ \mathsf{node}(pid') = n \end{array}}{[\langle e, pid, q, pl, b\rangle \parallel s_1, \mathsf{node}(pid), nq_1] \parallel [s_2, n, nq_2] \xrightarrow{\tau}}$$

$$[\langle e', pid, q, pl, b\rangle \parallel s_1, \mathsf{node}(pid), nq_1] \parallel [\langle f(v_1,\ldots,v_m), pid', \epsilon, \emptyset, false\rangle \parallel s_2, n, nq_2]$$

Figure 8: New spawn-rules (Symmetric spawn_link-rules omitted)

Everything defined in the original semantics works in the extended semantics, with a few exceptions. We have to change the *com*-rule (Fig. 3) slightly, so that it only applies in the situation where both processes are running on the same node. Further, we need to do a small (non-functional) modification in order to export extra information about the sender of a message to the layer we are adding. This is done by replacing the sending operator (!) with a tagged version ($!_{from}$), where *from* is the sender of the message. This change is straightforward and is applied to all the send operators in the single-node semantics. An example of the tagged send operator can be seen in the new *com*-rule presented in Fig. 7. The new *com*-rule has an added side condition, which restricts its application to the case when the sender and the receiver are running in the same Erlang system. Finally, to be able to spawn new processes in the multi-node setting, we need to refine the existing spawn-rule (we add a side condition, assuring that it is a local spawn) and add a distributed spawn-rule. (And similar changes for spawn_link.) The new ($spawn_{dist}$) and modified ($spawn_{local}$) spawn-rules are presented in Fig. 8. We should note that the distributed spawn rule is atomic, that is, the new process is created (but does not necessarily start executing) immediately at the remote node.

In addition to these changes, we also introduce a new way of writing an erlangDeadProcess (previously written $\langle pid,\ plm \rangle$), namely: $\langle\!|pid,\ plm|\!\rangle$.

4.5 Transitions

Multi-node systems transitions are labeled by actions. The actions that can occur at the level of nodes are defined below.

Definition 16 The Multi-node system actions, ranged over by $\gamma \in$ erlangMultiNodeSysAction are:

γ ::=	τ		silent action
|	$pid!_{from}sig$		output action
|	$pid?_{from}sig$		input action
|	die($node$)		node failure
|	disconnect($node1, node2$)		node disconnection

That is, the actions visible in erlangMultiNodeSysAction are only the node-to-node communication and node failure. Messages sent between processes executing on the same node are not visible at this level. Note also that at this level the input actions (?) are tagged with a *from*. This is not strictly necessary from a functionality point of view, but (as we see in Def. 22) *fairness* can be expressed in a simple and elegant way with tagged input actions.

Definition 17 The transistion relation for Erlang Multi-node systems, $\rightarrow$: erlangMultiNodeSystem $\times$ erlangMultiNodeSysAction $\times$ erlangMultiNodeSystem, written $n_1 \xrightarrow{\gamma} n_2$, is the least relation satisfying the rules in Fig. 9 – Fig. 14.

$$output_{node}\ \frac{s_1 \xrightarrow{pid\ !_{from} sig} s_1' \qquad \mathsf{node}(from) \neq \mathsf{node}(pid)}{[s_1, \mathsf{node}(from), nq_1] \parallel [s_2, \mathsf{node}(pid), nq_2] \xrightarrow{pid\ !_{from} sig} [s_1', \mathsf{node}(from), nq_1] \parallel [s_2, \mathsf{node}(pid), nq_2 \cdot (from, pid, sig)]}$$

$$output2_{node}\ \frac{s_1 \xrightarrow{pid\ !_{from} \mathsf{link}(from)} s_1' \qquad (pid, from, \mathsf{exited}(pid, \mathtt{noconn})) \notin nq}{[s_1, \mathsf{node}(from), nq] \parallel [\![\mathsf{node}(pid), npl]\!] \xrightarrow{from\ !_{pid} \mathsf{exited}(pid, \mathtt{noconn})} [s_1', \mathsf{node}(from), nq \cdot (pid, from, \mathsf{exited}(pid, \mathtt{noconn}))] \parallel [\![\mathsf{node}(pid), npl]\!]}$$

$$output3_{node}\ \frac{s_1 \xrightarrow{pid\ !_{from} sig} s_1' \qquad \begin{array}{r} sig \neq \mathsf{link}(from)\ \vee \\ (pid, from, \mathsf{exited}(pid, \mathtt{noconn})) \in nq_1 \end{array}}{[s_1, \mathsf{node}(from), nq_1] \parallel [\![\mathsf{node}(pid), npl]\!] \xrightarrow{\tau} [s_1', \mathsf{node}(from), nq_1] \parallel [\![\mathsf{node}(pid), npl]\!]}$$

Figure 9: Inter-node communication – Output rules

4.6 Operational output rules

In Fig. 9 the first rule, $output_{node}$ is the normal output rule. In this rule a message is sent to a process executing in a live ERTS, the message is appended to the node message queue nq. The message is later delivered to the receiving process by an input rule. The $output2_{node}$-rule generates an appropriate reply to a link-request made to a process on a dead node. A reply is only generated if there is not already a reply in the node queue of the sender (i.e. a message waiting for delivery in nq) for that particular process identifier. The reason to look inside nq for an exit-message is that as long as the error-message has not reached the linking process, it can not act upon the error. Therefore it can not know that it should establish a new link, and thus no new error-message should be constructed. The last output rule $output3_{node}$ take care of all other messages to a processes on a dead node, and simply discards them.

4.7 Operational input rules

In Fig. 10 there are two input-rules. The input rule $input_{node}$ uses the function nqMatch to retreive a message from the node message queue in a non-deterministic fashion. The selected message (sig) is then delivered to the actual receiving process. The nqMatch function is defined below. In Fig. 10 there is also the $silent_{node}$-rule, applied for everything except communication happening at system/process level. As with the output rules, note that messages sent between processes executing on the same node does not end up in the *node message queue*. They are handled by the modified *com*-rule (Fig. 7).

$$input_{node}\ \frac{s \xrightarrow{pid\ ?sig} s' \qquad \mathsf{nqMatch}(nq,from,pid) = sig}{[s, node, nq] \xrightarrow{pid\ ?_{from} sig} [s', node, nq \setminus (from,pid,sig)]}$$

$$silent_{node}\ \frac{s \xrightarrow{\tau} s'}{[s, node, nq] \xrightarrow{\tau} [s', node, nq]}$$

Figure 10: Inter-node communication – Input-rules

Definition 18 nqMatch(erlangNodeQueue,erlangPid,erlangPid), is a function that given an Erlang node message queue, a sender process id (*from*) and a receiver process id (*to*) returns the first message in the queue sent by *from* to *to*, e.g.

$$\begin{aligned} nq &= (a_2, b_1, c_1) \cdot (a_1, b_2, c_1) \cdot (a_1, b_2, c_2) \cdot (a_1, b_2, c_1) \\ &\Rightarrow \mathsf{nqMatch}(nq, a_1, b_2) = c_1 \end{aligned}$$

In Fig. 10 we should note that the rule $input_{node}$ can be applied in an arbitrary order for pairs of a sender and a receiver. This means that messages can (possibly) be reordered. However, at the same time this rule introduces another problem, namely that a certain (sender,receiver)-pair is never considered. That means that the delivery of some messages could potentially be delayed forever. The problem is that many properties can not be proved for such a non-fair situation, to deal with this problem we have to state a fairness rule (in Sect. 4.9).

4.8 Operational node rules

$$\begin{gathered} links = \{(pid, pid') \mid pid' \in \mathsf{getProcLinks}(pid),\ pid \in \mathsf{pids}(s)\} \cup \mathsf{links}(nq) \\ linkMsgs = \{(from, to, \mathsf{exited}(from,\texttt{noconnection}) \mid (from,to) \in links\} \\ node\text{-}failure\ \frac{n' = \mathsf{deliverMsgsToNq}(n,linkMsgs)}{n \parallel [s, node, nq] \xrightarrow{\mathsf{die}(node)} n' \parallel [\![node, \mathsf{pids}(s)]\!]} \end{gathered}$$

Figure 11: Node-failure rule

There are four operational node rules, *node-failure*, *node-(re)start*, *node-disconnect* and *node-interleave*. The node-failure rule is presented in Fig. 11. It looks quite complicated, but most of the complexity is due to bookkeeping. When a node fail, and because we do not have any external handling of links, we have to collect all links that are currently defined and produce proper exit-messages. To simplify the collection of links, we use two functions links and getProcLinks (defined below) to collect the links and a third function deliverMsgsToNq (also defined below) to deliver the exit-messages. The node-failure rule also create a *dead ERTS* with the same node name as the failing node. The dead node also contains a list of all processes previously running on the node. One thing to

$$\textit{node-start}\ \frac{}{[\![node, npl]\!] \xrightarrow{\tau} [init() \parallel \{\langle\!|pid, \{\}|\!\rangle \mid pid \in npl\}, node, \epsilon]}$$

Figure 12: Start node rule

$$\begin{aligned}
msg_{1\leftarrow 2} &= \{(from, to, sig) \mid sig \in \mathsf{nqMatchAll}(nq, to, from), \\
&\qquad to \in \mathsf{pids}(s_1),\ from \in \mathsf{pids}(s_2)\} \\
msg_{2\leftarrow 1} &= \{(from, to, sig) \mid sig \in \mathsf{nqMatchAll}(nq, to, from), \\
&\qquad to \in \mathsf{pids}(s_2),\ from \in \mathsf{pids}(s_1)\} \\
links_1 &= \mathsf{filterLinks}(\{(pid_1, pid_2) \mid pid_2 \in \mathsf{getProcLinks}(pid_1),\ pid_1 \in \mathsf{pids}(s_1)\} \\
&\qquad \cup \mathsf{links}(nq_1), n_1, n_2) \\
links_2 &= \mathsf{filterLinks}(\{(pid_1, pid_2) \mid pid_2 \in \mathsf{getProcLinks}(pid_1),\ pid_1 \in \mathsf{pids}(s_2)\} \\
&\qquad \cup \mathsf{links}(nq_2), n_2, n_1) \\
fail_1 &= \{(from,to,\mathsf{exited}(from,\mathtt{noconn})) \mid (from, to) \in links_1\} \\
fail_2 &= \{(from,to,\mathsf{exited}(from,\mathtt{noconn})) \mid (to, from) \in links_2\}
\end{aligned}$$

$$\textit{node-disc}\ \frac{}{[s_1, n_1, nq_1] \parallel [s_2, n_2, nq_2] \xrightarrow{\mathsf{disconnect}(n_1,n_2)} [s_1, n_1, (nq_1 \setminus msg_{1\leftarrow 2}) \cdot fail_2] \parallel [s_2, n_2, (nq_2 \setminus msg_{2\leftarrow 1}) \cdot fail_1]}$$

Figure 13: Node-disconnect rule

note here is that there can be at most one link between a pair of processes, and therefore we can safely add all link messages directly to the node queues without worrying about message order. Another thing that we should observe is that links are collected both from the individual processes and the node message queue *nq*. The intuition behind this is that as soon as a process on another node has sent a link request, the sending process believes that it has a working link to the linked process. The linking mechanism is further discussed in Sect. 6.

Definition 19 The function links(erlangNodeQueue) traverses an Erlang node message queue and collects all pending link-request from the node queue. The function getProcLinks(erlangPid) return the *pl*-list (i.e. the list of linked nodes) for a process, given the process identifier of that process. Finally, the function deliverMsgsToNq(EMNS,Messages) deliver all messages to the correct node queue. E.g. let $n = [s_{11}, n_1, nq_1] \parallel [s_{21}, n_2, nq_2]$ and $p_1, p_3 \in \mathsf{pids}(s_{11})$ and $p_7 \in \mathsf{pids}(s_{21})$, then: $\mathsf{deliverMsgsToNq}(n, \{(p_7, p_1, sig_1), (p_3, p_7, sig_2)\}) =$
$[s_{11}, n_1, nq_1 \cdot (p_7, p_1, sig_1)] \parallel [s_{21}, n_2, nq_2 \cdot (p_3, p_7, sig_2)]$

In Fig. 12 the node-(re)start rule is presented. The interesting thing to notice here is that we create an erlangDeadProcess for each pid that has previously been running on the node (i.e. the process identifiers in *npl*). The reason for this is to simplify link handling. If all previous processes exists on the node, future link-requests sent to these processes get the correct respose without having to create any further semantic rules. The function init() is an initialization process which is started on a new node. What the init()-process does is not further specified in

$$interleave_{node}\ \frac{n_1 \xrightarrow{\gamma} n_1'}{n_1 \parallel n_2 \xrightarrow{\gamma} n_1' \parallel n_2}$$

Figure 14: Node interleaving (symmetrical rule omitted)

the semantics, but it could be thought of as any reasonable, and changeable from the outside, starting action for an Erlang node. For example starting a certain set of processes, or initiate some other chain of events.

The third node rule, node-disconnect, is presented in Fig. 13. The rule handles the situation when the communication channel between two nodes break down. The result of this is that all messages from/to the disconnected nodes that are currently in the node queue are lost, and that a set of link messages (exit-messages) are created and sent. The rule uses two functions nqMatchAll and filterLinks (defined below) to collect messages that are lost and links that should be converted to a exit-messages. The rule discards messages and adds link notifications to the node queues of the disconnected nodes. Some things should be noted, first, the order of messages between a pair of processes is not affected by this rule. Either the messages are delivered in order or not at all. Second, to drop all messages in transit between the disconnected nodes is a design choice, we could just as well drop only an arbitrary set of messages. This is further discussed in Sect. 6. Finally we note that there is no *node-(re)connect*-rule, since the reconnection of nodes is transparent at the semantic level.

Definition 20 The function nqMatchAll(erlangNodeQueue, To, From) is a function that given an Erlang node message queue, a sender process id (*from*) and a receiver process id (*to*) returns all messages in the queue sent by *from* to *to*. The function filterLinks($\mathcal{P}$(erlangPid $\times$ erlangPid), erlangNodeName, erlangNodeName) is a function that filters a set of process identifier pairs with respect to the node the processes are running at. E.g. let $\mathsf{node}(p_{1x}) = n_1$, $\mathsf{node}(p_{2x}) = n_2$ and $\mathsf{node}(p_{3x}) = n_3$ then: $\mathsf{filterLinks}(\{(p_{11}, p_{21}), (p_{12}, p_{34}), (p_{27}, p_{12})\}, n_1, n_2) = \{(p_{11}, p_{21})\}$

We should also take a closer look at what happens with the node-to-node communication when the receiving process terminates. When a process terminates, its message queue q disappears. That is all messages which have already been delivered to the process are deleted. If the rule $input_{node}$ is applied for a terminated process, i.e. if we deliver a message to a terminated process, this is handled by the rules in Table 3.17 in Fredlunds semantics [15]. That is, the underlying semantics properly destroy messages and reply to link-requests.

4.9 Fairness

As we noted above, the input-rule, i.e. the $input_{node}$ rule in Fig. 10, can be applied in such a way that some messages are never delivered. That is the rules themselves does not ensure that messages are delivered in a fair manner. This

is generally a bad thing, since many properties can not be proved in a non-fair system. Therefore we need to define a fairness-rule which will exclude certain unwanted behavior of the system. Fairness is defined in terms of permissable *execution sequences*.

Definition 21 An *execution sequence* is a sequence of Erlang Multi-node Systems n_i, together with corresponding Erlang Multi-node system actions γ_i written:

$$n_0 \xrightarrow{\gamma_0} n_1 \xrightarrow{\gamma_1} n_2 \xrightarrow{\gamma_2} \ldots.$$

Definition 22 [**Fairness for inter-node communication**]
It should hold for all execution sequences, $(\vec{n}, \vec{\gamma})$:

$$\forall i.\Big\{ n_i \xrightarrow{pid!_{from}sig} n_{i+1} \Rightarrow \exists j > i.\Big(n_j \xrightarrow{pid?_{from}sig} n_{j+1} \vee n_j \xrightarrow{die(\mathsf{node}(pid))} n_{j+1} \vee n_j \xrightarrow{\mathsf{disconnect}(\mathsf{node}(pid),\mathsf{node}(from))} n_{j+1} \vee n_j \xrightarrow{\mathsf{disconnect}(\mathsf{node}(from),\mathsf{node}(pid))} n_{j+1} \Big)\Big\}$$

That is, Definition 22 state that every sent message is eventually delivered to the recieving process, or the node where the receiving process is executed dies, or a node disconnection involving the sending and the receiving node occurs.

4.10 Message reordering

The motivation for extending Fredlund's single-node semantics was partly to capture the distributed behavior where messages were reordered. Therefore, we repeat the 'hello world' example from Sect. 3, but now we execute the program from Fig. 4 in the extended semantics. The example is presented in Fig. 15.

4.11 Node disconnection

Another part of the motivation for the multi-node semantics was to capture the behavior where nodes disconnect. Therefore, we conclude the presentation of the extended semantics with an example with node disconnection. The example is presented in Fig. 17, where we execute the program in Fig. 16 in the multi-node semantics. The program consists of two processes where the first process sends a the number sequence [1, 2, 3] to the other. The effect of the node disconnect is that the second process can receive the sequence [1, 3].

1. Initial system:

$N_0 = [\langle \texttt{PidC} = \texttt{spawn(nodeC,procC,[])} \ \ldots, p_0, \epsilon\rangle, n_0, \epsilon]$

2. The only scheduler option is to spawn `procC`. After that, the only option is to spawn `procB` since the `receive` in `procC` blocks. This results in three processes:

$N_0 = [\langle \texttt{PidC ! hello} \ldots, p_{01}, \epsilon\rangle, n_0, \epsilon]$
$N_1 = [\langle \texttt{receive } X \rightarrow \texttt{PidC ! } X \texttt{ end}, p_{11}, \epsilon\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon\rangle, n_2, \epsilon]$

3. Only N_0 can make progress, since N_1 and N_2 are blocked on a `recieve` statement:

$N_0 = [\langle \texttt{PidB ! world}, p_{01}, \epsilon\rangle, n_0, \epsilon]$
$N_1 = [\langle \texttt{receive } X \rightarrow \texttt{PidC ! } X \texttt{ end}, p_{11}, \epsilon\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon\rangle, n_2, \epsilon \cdot (p_{01}, p_{21}, \texttt{hello})]$

4. Now we have two options, either apply the $input_{node}$ on N_2 or let N_0 send its second message. Since the purpose is to show message re-ordering we let N_0 proceed:

$N_0 = [\langle p_{01}, \epsilon \rangle, n_0, \epsilon]$
$N_1 = [\langle \texttt{receive } X \rightarrow \texttt{PidC ! } X \texttt{ end}, p_{11}, \epsilon\rangle,$
$\quad n_1, \epsilon \cdot (p_{01}, p_{11}, \texttt{world})]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon\rangle,$
$\quad n_2, \epsilon \cdot (p_{01}, p_{21}, \texttt{hello})]$

5. Again we have two options, we can apply $input_{node}$ to either N_1 or N_2, to illustrate our point, we choose N_1:

$N_1 = [\langle \texttt{receive } X \rightarrow \texttt{PidC ! } X \texttt{ end}, p_{11}, \epsilon \cdot \texttt{world}\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon\rangle, n_2, \epsilon \cdot (p_{01}, p_{21}, \texttt{hello})]$

6. Now we continue to ignore N_2 and let p_{11} read its message and send it to p_{21}:

$N_1 = [\langle p_{11}, \epsilon \rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon\rangle,$
$\quad n_2, \epsilon \cdot (p_{01}, p_{21}, \texttt{hello}) \cdot (p_{11}, p_{21}, \texttt{world})]$

7. Finally we have arrived in the wanted situation, we can apply $input_{node}$ to N_2, and by its construction it is perfectly ok to deliver the `world` message:

$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon \cdot \texttt{world}\rangle,$
$\quad n_2, \epsilon \cdot (p_{01}, p_{21}, \texttt{hello})]$

8. Now `world` is received before `hello`.

Figure 15: Hello World – Multi-Node Execution

5 Properties of the Multi-Node Semantics

In the previous section we have defined a distributed semantics for a subset of Erlang. The limitations are the same as Fredlund have in the single node semantics [15]; in short standard Erlang without: modules, floats, references, binaries, ports, the catch-expression and some internal process state information such as process dictionaries. In the design of the semantics we have had several informal properties which we have intended for the distributed semantics. In this section we argue for some of them and describe why they are desirable and why they actually hold for the distributed semantics.

```
init() ->
    PidB = spawn(?NODE2,?MODULE,procB,[]),
    PidA = spawn(?NODE1,?MODULE,procA,[PidB]).

procA(PidB) ->
    PidB ! 1,
    PidB ! 2,
    PidB ! 3.

procB() ->
    receive X -> ok end,
    receive Y -> ok end,
    receive Z -> ok end.
```

Figure 16: Simple One-Two-Three counting program

5.1 Extension

The first property we want is that the distributed semantics is a true extension of the single-node semantics. That is if we take a single-node system and execute it in the multi-node semantics (this is possible with only minor modifications of the system) it should work exactly the same. In order to express 'exactly the same' and 'minor modifications' in a strict way we have to define the mkDist function as well as an *execution sequence*.

Definition 23 The function mkDist(erlangNodeId,erlangSystem) takes a single-node Erlang system and prepare it for execution in the multi-node semantics (on the given node). The necessary change is to replace each $\mathsf{spawn}(f, [v_1, \ldots, v_m])$ with the distributed variant $\mathsf{spawn}(n, f, [v_1, \ldots, v_m])$, where n is the given node.

Definition 24 An *execution sequence* for an Erlang is defined to be the sequence of system actions (see Def. 9) performed by the executed system. Since the scheduler is non-deterministic an Erlang system will have a (large) set of possible execution sequences.

Prop 1. A single-node Erlang system ($s \in \mathsf{erlangSystem}$) which is prepared for execution in the multi-node semantics by the mkDist function has exactly the same set of possible *execution sequences* as the system s executed in the single-node semantics, as long as the node is not allowed to fail.

This is true because the only semantic rules that are invoked are those in the single node semantics (together with the non-interfering $silent_{node}$ rule). This we can be sure of because there is only one way to 'escape to' the multi-node semantics, namely by communication not caught by the *com*-rule (Fig. 7). And because we have the side condition in the $spawn_{local}$-rule that the node is the same for the spawned processes, all communication is caught by the *com*-rule. Note that we have to disallow the node failure, or the distributed version of the system

1. Initial system:

   $N_0 = [\langle \texttt{PidB} = \texttt{spawn}(\texttt{n}_2,\text{procB},[])\ \ldots, p_0, \epsilon\rangle, n_0, \epsilon]$

2. The only scheduler option is to spawn `procB`. After that, the only option is to spawn `procA` since the `receive` in `procB` blocks. This results in two processes (the first one is terminated):

   $N_1 = [\langle \texttt{PidA} \,!\, \texttt{1} \ldots, p_{11}, \epsilon\rangle, n_1, \epsilon]$
   $N_2 = [\langle \texttt{receive}\ X \rightarrow \texttt{ok}\ \texttt{end} \ldots, p_{21}, \epsilon\rangle, n_2, \epsilon]$

3. Again there is only one option, to let `PidA` send `1`:

   $N_1 = [\langle \texttt{PidA} \,!\, \texttt{2} \ldots, p_{11}, \epsilon\rangle, n_1, \epsilon]$
   $N_2 = [\langle \texttt{receive}\ X \rightarrow \texttt{ok}\ \texttt{end} \ldots, p_{21}, \epsilon\rangle, n_2, \epsilon \cdot (p_{11}, p_{21}, \texttt{1})]$

4. To illustrate our point, we now let N_2 execute, the $input_{node}$- rule is applied:

   $N_1 = [\langle \texttt{PidA} \,!\, \texttt{2} \ldots, p_{11}, \epsilon\rangle, n_1, \epsilon]$
   $N_2 = [\langle \texttt{receive}\ X \rightarrow \texttt{ok}\ \texttt{end} \ldots, p_{21}, \epsilon \cdot \texttt{1}\rangle, n_2, \epsilon]$

5. Now we proceed by letting N_1 execute and send another number:

   $N_1 = [\langle \texttt{PidA} \,!\, \texttt{3} \ldots, p_{11}, \epsilon\rangle, n_1, \epsilon]$
   $N_2 = [\langle \texttt{receive}\ X \rightarrow \texttt{ok}\ \texttt{end} \ldots, p_{21}, \epsilon \cdot \texttt{1}\rangle, n_2, \epsilon \cdot (p_{11}, p_{21}, \texttt{2})]$

6. Now assume that the nodes disconnect, and thus we apply the *node-disconnect* rule, note that the message already delivered to p_{21} is not affected:

   $N_1 = [\langle \texttt{PidA} \,!\, \texttt{3} \ldots, p_{11}, \epsilon\rangle, n_1, \epsilon]$
   $N_2 = [\langle \texttt{receive}\ X \rightarrow \texttt{ok}\ \texttt{end} \ldots, p_{21}, \epsilon \cdot \texttt{1}\rangle, n_2, \epsilon]$

7. Now the nodes are re-connected, and we can proceed by letting N_1 execute and send yet another number:

   $N_1 = [\langle p_{11}, \epsilon\rangle, n_1, \epsilon]$
   $N_2 = [\langle \texttt{receive}\ X \rightarrow \texttt{ok}\ \texttt{end} \ldots, p_{21}, \epsilon \cdot \texttt{1}\rangle, n_2, \epsilon \cdot (p_{11}, p_{21}, \texttt{3})]$

8. Now we see that p_{21} will receive the sequence [1,3], which is not possible in the single-node semantics.

Figure 17: One-Two-Three – Multi-Node Execution

would have a larger set of possible execution sequences. Node disconnect is not a problem since we have only one node!

5.2 Message Reordering and Node Disconnect

The main motivation for the semantics for distributed Erlang was to be able to express message reordering and node disconnect in the semantics, it is therefore desirable that this is indeed possible.

Prop 2. In a distributed system, a message sent from a process (A) via a second process (B) to a third process (C), can arrive before a message sent directly from A to C at some earlier point in time. Messages between two processes at different nodes can also be lost due to node disconnection.

In the examples in Fig. 15 and Fig. 17 we demonstrate that messages can arrive

in the desired way and that node disconnection is indeed possible. By the design of the function nqMatch the opposite, namely that messages between a pair of processes are always ordered, is also ensured.

5.3 Expressiveness

It is also important that the extended semantics is complete in terms of expressiveness. Every correct distributed system must be able to execute in the semantics, and it must not get stuck because there is something not expressible in the semantics.

Definition 25 Progress for a process is equivalent to applying one of the evaluation rules for processes. Further an Erlang system can make progress if and only if any of its processes can make progress.

Prop 3. An Erlang (Multi-node) system which is not in a dead-locked situation and that does have at least one live process should be able to make progress according to the semantic rules.

This is a property that should hold for the underlying semantics, even though it is not proved by Fredlund in [15]. We would like to prove something similar for ERTS/EMNS, but here we have the extra complications of nodes. Because of the rules *node-failure*,*node-start* and *node-disconnect* an EMNS can always do something namely that a node fail or restart or that two nodes disconnect.

5.4 Finite systems stays finite

Another concern is that the extension of the semantics may introduce unwanted overhead in terms of resources. Since the semantics should model the behavior of the run-time system it is important that a system that executes with finite bounds on the message queues does so also in the semantics.

Prop 4. A finite system (that is with finite bounds of the length of the message queues) in the single-node semantics is still finite in the multi-node semantics.

The main argument for why this holds is that there is a one-to-one mapping of messages. That is there is no place in the multi-node semantics where a message is duplicated, therefore the number of messages stays the same. Since we also have the fairness rule all messages should eventually be delivered, and there is no inherent way that systems become infinite in the distributed semantics.

5.5 A word of caution

After we have argued above that the desired properties of the distributed semantics holds it is important to note a few artifacts of the functional differences between a single-node and a multi-node system. Because of the functional differences, a non-deadlocking single-node system might very well dead-lock if run

in a distributed setting. For example imagine a system which depends on the message ordering implied by the single-node semantics[§], such a system could easily dead-lock if we distribute it over several run-time systems. In the same way, also a finite single-node system might very well become infinite if it is run in a distributed environment.

6 Discussion

The fundamental characteristics of Erlang are described by Armstrong in his thesis [1]. Armstrong describes how the concept of *concurrency oriented programming* led to the development of Erlang. The original thoughts on distribution are further described by Wikström [23]. In a concurrency oriented programming language the following is specified for *message passing*: "Message passing between a pair of processes is assumed to be ordered."

This is indeed true for the semantics presented by Fredlund [15], but due to the construction of the *com*-rule (in Fig. 3), even stronger properties hold in the single-node semantics. In Fredlund's single-node semantics the delivery of a message to another process is instantaneous, meaning that **all** messages are delivered in exactly the order they are sent. Because of how the standard (OTP) Erlang run-time system is implemented, this happens to be true also for a real Erlang system where all processes are running on the same node. However, it is not true in general for a concurrency oriented programming language, and specifically not in a distributed setting with several different Erlang nodes.

There is an intricate choice here, on one side we have the de facto standard, the OTP implementation of the Erlang run-time system where local communication is instantaneous. On the other side we have the different Erlang specifications, where no support for such instantaneous communication can be found. Our original concern was to produce a semantics to be used for model checking, and therefore the presentation is biased by this. Since Fredlund's single-node semantics faithfully describes what actually happens inside the standard run-time system, we argue that for efficient model checking of Erlang systems (to be run on in the standard run-time system), the underlying semantics should be kept as it is. Fredlund's intra-node message passing is not consistent with the Erlang specifications, but using a special version of local message passing makes certain (local) systems easier to reason about.

Nevertheless, it is somewhat unconventional to produce a semantics for a particular implementation of a language, and thus one could argue that we should instead present the more general semantics. The alternative then is to only have the kind of message passing rules that we have in the node-to-node communication. Such a modification would be rather straightforward to do. The consequence of this is an overall simpler (and more general) semantics, which is less restrictive for a local system. However, this is problematic in the model checking context,

[§]One could indeed say that such a system is simply containing a bug, since the enforcment of such a static message order is nowhere to be found in the Erlang specifications. Nevertheless, since the de facto standard, the OTP Erlang run-time system implementation, actually behave this way such programs are going to be written.

since it results in a bigger state space. Another problem is the introduction of false negatives, because a local system might appear to fail due to an order of events not possible in reality (in the standard Erlang implementation). Such a general semantics would however be useful in case of a future Erlang implementation that adheres more closely to the Erlang specifications. (Such an implementation would of course also require changes to model checkers using the semantics presented in this paper.)

This picture may also be further complicated in the future by the introduction of multi-core systems and the SMP-version of the Erlang run-time system.

In the development of the multi-node semantics, we have also made several other distinctive choices. One particular choice is seen in the *node-disconnect* rule, where all messages currently in transit between two nodes are dropped. We could just as well have dropped an arbitrary sequence of messages in the node queue. However, dropping all messages is certainly easier and it makes the fairness rule less complicated. This is also an example of where a simpler rule offer the same expressivity as a more complicated one. Every sequence of messages in the node queue could indeed be dropped, it is just a matter of applying the rule at the right time. Another design choice we made was to introduce one message queue for 'messages in transit' per node. There is no functional motivation behind this choice, we could just as well have settled for one single global message queue, but in the end we thought it to be more aesthetic to have one queue per node.

Quite many of the rules presented in Section 4 handle the link-messages. The link mechanism is a very useful construction and many distributed implementations rely on this functionality. It is important to observe that we must treat links differently from ordinary messages in order to faithfully describe Erlang programs. For example, take a look at the Erlang program in Fig. 18. If we run `procA` it should be possible to sometimes trap the exit message (i.e. get an {`'EXIT',pid,kill`} from `procB` and sometimes just get a {`'EXIT',pid,noproc`} back, indicating that process B had already terminated. This behavior can be observed by running the program repeatedly. Although the result is heavily dependent on machine load and network load, with 1000 runs, almost everytime both behaviors could be observed. This means that it would be incorrect to treat the link-message as ordinary message, since the message order between a pair of processes is respected and then an order of events such as getting {`'EXIT',pid,kill`} from process B would be impossible.

In Fredlund's single-node semantics, (and here seen in Fig. 2) a separation is made between link-messages and other messages, which ensures the correct behavior. However, when dead nodes are involved, some special care is needed, which results in special link-rules as seen in Fig. 9.

Another part of the linking mechanism are the somewhat complicated *node-failure* and *node-disconnect* rules (Fig. 11 and Fig. 13), where we have to collect link messages from the node message queue. This is because we are modelling the link mechanism in a different way from the actual Erlang implementation. This actually an important observation in a more general perspective. The goal with the semantics is to be able to express all the possible behavior of the Erlang implementation, and not to describe how the implementation actually works. In

```
procA() ->
    PidB = spawn(?ANOTHERNODE,?MODULE,procB,[]),
    PidB ! a,
    process_flag(trap_exit,true),
    link(PidB),
    ...

procB() ->
    receive a ->
    exit(kill)
    end.
```

Figure 18: Erlang program - Linking

the Erlang implementation, the run-time system keeps track of links via a timeout construction. In the semantics we instead do the book keeping (so to say) at the other end. Therefore, we have to take extra care when node fails since messages in *nq* are otherwise lost.

The *node-disconnect* rule looks rather horrible from a programmers point of view, at any time, all messages between a pair of nodes may be lost. However this is not as disastrous as one might think, as long as one is aware that this might happen. This is because the link-mechanism works in the node disconnect case, and as long as communication is restricted to monitored receivers there is no immediate danger.

Finally we should discuss one limitation of the distributed semantics, namely that *monitors* are not a part of the semantics. This is a limitation because monitors are widely used, and the correctness of many distributed Erlang systems rely on monitors. One is tempted to belive that it is possible to implement monitors in terms of links. This is however only partly true, since monitors would have to be implemented using a named and dedicated process for each node. This means that in order to get a correct behavior we have to ensure that no one is for example killing the monitor process. Therefore, it is not obvious how to implement monitors in terms of links and it seems that we have to make some non-trivial assumptions.

7 Related Work

The semantics for Erlang is informally described in [2]. Thereafter, a first, not completed, attempt to formally specify the semantics of Erlang was made by Petterson [20]. Petterson, inspired by similar work with Standard ML and Relational ML, used Natural Semantics but did not finish the work. Following this, the Formal Design Techniques group at the Swedish Institute of Computing Science (SICS) developed a number of formal (operational) semantics for different subsets of Erlang, for example [12] and [13]. These attempts, compared to the semantics presented by Fredlund in [15], are not as direct and lacks the clear separation between the functional and the concurrent part of the semantics. A

completely different approach is taken by Huch in [19]. Huch present a semantics for (a smaller subset) of Erlang, which is more direct and relies heavily on contextual information. All these approaches except Petterson's consider systems which are not fully distributed since they do not deal with nodes.

Both [12] and [19] make use of subsets of Erlang referred to as core fragments of Erlang. These references should not be confused with the Core Erlang project [10], which defines a complete (with respect to representing all possible Erlang programs) core fragment of Erlang. Core Erlang is in the Erlang compiler used as the intermediate format where optimizations and transformations are applied, therefore its use is mostly syntactic. For Core Erlang the semantics is given in a structured but also informal way, and does not directly speak about nodes or message delivery.

The distributed semantics for Erlang presented in [11] has been used in a model checker for Erlang, McErlang [16]. Implementing a model checker is the ultimate test for a semantics, and several limitations were indeed found, which motivated the further work on a distributed semantics for Erlang.

8 Conclusions and Future Work

In 2005, we proposed an extension of the Fredlund's single-node semantics into a distributed (multi-node) semantics for Erlang [11]. We augmented the single-node semantics with a distributed layer, and were able to successfully model multi-node programs in the extended semantics. Together with the distributed semantics there was also a warning to the community of potential pit-falls with testing and verification using a single-node semantics.

Later the multi-node semantics was used in the implementation of a model checker for Erlang (McErlang, [16]). The model checker implementation revealed some inconsistencies as well as a major shortcoming in the multi-node semantics. The main problem was the Erlang behavior in the case of *node disconnect*. Two Erlang nodes can become disconnected from each other if the link between them fails, when this happens both involved nodes regard the other node as dead. This does have some interesting consequences when the nodes later re-connect. The correct behavior was later implemented in the model checker, however this behavior could not be modeled in the multi-node semantics we had proposed.

In this new presentation of a distributed semantics for Erlang, we have re-structured some parts of the distributed layer. Further, we have added the node disconnect functionality, we have corrected errors in the original presentation and we have simplified and clarified the presentation in many aspects. The result is a more accurate and more expressive multi-node semantics for Erlang, with a clearer presentation and less complicated semantic rules. We have also added a discussion of the desired properties of the multi-node semantics.

Future Work The introduction of multi-core computer systems and the development of a SMP-version of the Erlang run-time is already a fact. And it will be interesting to see if there are any new semantic implications because of this. There is also further work to be done with model checking Erlang in the distributed setting.

Acknowledgements

We thank Koen Claessen for his valuable comments on earlier versions of this paper.

References

[1] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, December 2003.

[2] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1996.

[3] T. Arts, C. Benac Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *Int. J. on Software Tools for Technology Transfer*, 5(2-3):205–220, 2004.

[4] T. Arts, C. Benac Earle, and J. J. Sánchez Penas. Translating Erlang to mCRL. In *Fourth International Conference on Application of Concurrency to System Design*, pages 135–144, Hamilton (Ontario), Canada, June 2004. IEEE computer society.

[5] T. Arts, K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. In *Lecture Notes in Computer Science*, volume 3395, pages 140 – 154. Springer, Feb 2005.

[6] T. Arts and L.-Å. Fredlund. Trace analysis of Erlang programs. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 16–23. ACM Press, 2002.

[7] T. Arts and J. Hughes. Erlang/QuickCheck. In *Ninth International Erlang/OTP User Conference*, Nov. 2003.

[8] J. Barklund and R. Virding. Erlang 4.7.3 reference manual. Draft (0.7), Ericsson Computer Science Laboratory, 1999.

[9] J. Blom and B. Jonsson. Automated test generation for industrial Erlang applications. In *ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 8–14, New York, NY, USA, 2003. ACM Press.

[10] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S. Nystrm, M. Petterson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 2000-30, Department of Information Technology, Uppsala University, November 2000.

[11] K. Claessen and H. Svensson. A semantics for distributed erlang. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 78–87, New York, NY, USA, 2005. ACM Press.

[12] M. Dam and L.-Å. Fredlund. On the verification of open distributed systems. In *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing*, pages 532–540, New York, NY, USA, 1998. ACM Press.

[13] M. Dam, L.-Å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 150–185, London, UK, 1998. Springer-Verlag.

[14] L.-Å. Fredlund. Towards a semantics for Erlang. In *Foundations of Mobile Computation: A Post-Conference Satellite Workshop of FST & TCS 99*, Institute of Mathematical Sciences, Chennai, India, Dec 1999.

[15] L.-Å. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

[16] L.-Å. Fredlund and C. B. Earle. Model checking erlang programs: the functional approach. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 11–19, New York, NY, USA, 2006. ACM Press.

[17] L.-Å. Fredlund, D. Gurov, and T. Noll. Semi-automated verification of Erlang code. In *ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 319, Washington, DC, USA, 2001. IEEE Computer Society.

[18] L.-Å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):405 – 420, Aug 2003.

[19] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 261–272, New York, NY, USA, 1999. ACM Press.

[20] M. Petterson. A definition of Erlang (draft). Manuscript, Department of Computer and Information Science, Linköping University, 1996.

[21] M. Widera. Flow graphs for testing sequential Erlang programs. In *ERLANG '04: Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*, pages 48–53, New York, NY, USA, 2004. ACM Press.

[22] U. Wiger. Fault tolerant leader election. http://www.erlang.org/.

[23] C. Wikstrom. Distributed programming in Erlang. In *PASCO'94, First International Symposium on Parallel Symbolic Computation*, Linz, Austria, Dec 1994.

Paper 4

McErlang: A Model Checker for a Distributed Functional Programming Language

This paper was written together with Lars-Åke Fredlund for the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007), in Freiburg, Germany, October 2007. The paper included here has a few minor corrections and is also typeset in a different style.

McErlang: A Model Checker for a Distributed Functional Programming Language*

Lars-Åke Fredlund[†1], and Hans Svensson[2]

[1] Facultad de Informática, Universidad Politécnica de Madrid, Spain
fred@babel.ls.fi.upm.es

[2] Chalmers University of Technology, Göteborg, Sweden
hanssv@cs.chalmers.se

Abstract

We present a model checker for verifying distributed programs written in the Erlang programming language. Providing a model checker for Erlang is especially rewarding since the language is by now being seen as a very capable platform for developing industrial strength distributed applications with excellent failure tolerance characteristics. In contrast to most other Erlang verification attempts, we provide support for a very substantial part of the language. The model checker has full Erlang data type support, support for general process communication, node semantics (inter-process behave subtly different from intra-process communication), fault detection and fault tolerance through process linking, and can verify programs written using the OTP Erlang component library (used by most modern Erlang programs).

As the model checking tool is itself implemented in Erlang we benefit from the advantages that a (dynamically typed) functional programming language offers: easy prototyping and experimentation with new verification algorithms, rich executable models that use complex data structures directly programmed in Erlang, the ability to treat executable models interchangeably as programs (to be executed directly by the Erlang interpreter) and data, and not least the possibility to cleanly structure and to cleanly combine various verification sub-tasks. In the paper we discuss the design of the tool and provide early indications on its performance.

Categories and Subject Descriptors: D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking

General Terms: Verification

†The author was supported by a Ramón y Cajal grant from the Spanish Ministerio de Educación y Ciencia, and the DESAFIOS (TIN2006-15660-C02-02) and PROMESAS (S-0505/TIC/0407) projects.

1 Introduction

To model check a modern distributed functional programming language is by no means a small task and there are many design decisions that have to be taken. One of the largest decisions is to choose between: (1) translating the program into some existing formalism and use (or possibly extend) existing model checking tools for this formalism, or (2) implement the verification algorithms directly for the language to be model checked. (In fact there is also a third alternative, namely to translate into a formalism which has been constructed for this particular task and implement tools for this formalism. This approach is briefly discussed in section 7.)

There are advantages (as well as weaknesses) with both approaches; Existing tools are probably optimized, and thus efficient to use. Translating into an existing formalism means that everything in the language has to be modeled, including data. Finding a suitable formalism might not be easy. Implementing model checking algorithms efficiently is hard and time consuming. Having the model checker in the language itself means that the pure functional part can be handled in a simple and efficient way.

One existing model checking tool for Erlang is the `etomcrl` tool set [8], which consists of a translator from Erlang to μ`CRL`, a state space generator for μ`CRL` specifications, and the `CADP` state space analysis tools. Thus it is an example of the first alternative above, however early in the development of `etomcrl` its principal authors (Thomas Arts and Clara Benac Earle) were thinking of an implementation in Erlang itself. In the end the Erlang route was rejected because it was thought that it would be more efficient to reuse existing tools.

In this paper we describe the development and implementation of the McErlang model checker which follows the second implementation alternative above. The development of McErlang was started for several reasons. One reason was the curiosity to find out just how well an implementation in Erlang would work in practice. The main reason however was the wish to model check distributed as well as fault tolerant Erlang programs. (Both distribution and fault tolerance are missing in the `etomcrl` tool set). It was deemed too hard to extend the `etomcrl` tool set with the concepts of distribution and fault tolerance. The importance of supporting the distributed parts of Erlang is illustrated by Claessen and Svensson [11]. In their paper they show that it easy to overlook errors due to the loose synchronisation between processes in the distributed setting. They also demonstrate the presence of such errors in an open source Erlang implementation.

One significant advantage that the implementation in Erlang itself brings is that we can model check a larger fragment of the language than is normally achievable. There is for instance no separate step to compile the data sub-language of the source specification language to the often restrictive data language available in a model checker. Instead supporting the (purely functional) data part of Erlang is completely trivial; we can simply reuse the existing Erlang run-time system unchanged. It is in our opinion crucial to support a large fragment of Erlang in order to achieve some measure of acceptance of our tool by Erlang programmers.

The Erlang language contains many features not found in most normal pro-

gramming languages (unless add-on libraries are used): dynamic types (i.e., no static type system), concurrency via a process concept, inter-process communication using only asynchronous message passing, distribution by mapping processes onto (remote) processing nodes, fault tolerance via a failure detector mechanism, and a standardized set of high-level components built on top of this foundation. As Erlang programmers frequently make use of all these features we think it is vital that the verification tool supports them too.

Nevertheless, by choosing to implement a model checker in a functional programming language we risk paying a price with regards to loss of execution performance and increased storage requirements; there is clearly a trade-off between easy experimentation and expressive power on one hand, and implementation efficiency on the other. With the McErlang tool we want to explore this trade-off, and we hope that by not having to simulate the functional parts of Erlang this model checking approach is rather efficient.

Since we had access to a prototype implementation of the distributed Erlang semantics in Haskell, we also did some experiments with implementing a model checker for Erlang in Haskell. We did not implement a full model checker, but the experiments gave us some insight in the strengths and weaknesses with such an approach. In section 7 we briefly discuss these results to get a different perspective on the McErlang implementation.

One of the design goals with McErlang was that it should be easy to use. All that is needed to use McErlang is a program to test, a specification of the environmental constraints and a property to check. All three are written entirely in Erlang. The environmental constraints describe how the program is executed in the implementation of the run-time system provided by McErlang. The property is given in the form of a monitor/automaton that is executed in parallel with the program, checking for errors along the execution path. This work flow is discussed further in section 4 and section 5 and is illustrated in Fig. 7.

Contributions The main contribution of the paper is a presentation of the tool McErlang. The paper is not so much about the theory behind model checking or the semantics of Erlang, instead we focus on design choices, implementation decisions, adaptability and usability. McErlang is a model checker for Erlang implemented in Erlang, it supports a large subset of the Erlang programming language. In particular it supports all of the distributed and fault-tolerant parts of Erlang. This is especially important since distributed and fault-tolerant implementations are known to be error prone and hard to test and debug. McErlang is also easy to use and should be accessible to an ordinary Erlang programmer. Finally McErlang is designed in a very modular way and can easily be adapted to support other target languages.

Paper organization The next section contains an introduction to the most important features of the Erlang programming language and section 3 contains a description of the most prominent features of the Erlang semantics. In section 4 the parametric design of McErlang is described and section 5 presents the model checker itself, i.e., essentially an on-the-fly model checker which executes Büchi

automatons (coded in Erlang) in parallel with the Erlang program under study. In section 6 we show some results and examples of using the model checker. Section 7 discuss a number of design choices in more detail, and section 8 summarises related work. Finally section 9 draws conclusions, and outline future research work.

Download McErlang can be downloaded at the following location: `http://babel.ls.fi.upm.es/~fred/McErlang/`.

2 The Erlang Programming Language

Erlang is a programming language developed at Ericsson for implementing telecommunication systems [4, 2]. It provides a functional sub-language, enriched with constructs for dealing with side effects such as process creation and inter–process communication via message passing. Moreover Erlang has support for writing distributed programs; *processes* can be distributed over physically separated processing *nodes*.

Today several commercially available products developed by Ericsson and other companies are at least partly programmed in Erlang, an example is the AXD 301 ATM switch [10]. The software of such products is typically organized into many, relatively small, source modules, which at run-time execute as a dynamically varying number of processes operating in parallel and communicating through asynchronous message passing. The highly concurrent and dynamic nature of such software makes it particularly hard to debug and test.

Erlang programmers, of course, mostly work with ready-made higher-level language components rather than the basic language. In practice programmers predominantly use the OTP component library [28], which offers a number of useful software components such as: a generic server component for client-server communication, a finite-state machine component, and a supervisor component that restarts failed processes. Our approach to model checking Erlang programs can verify software that is built using both the core message passing language and with these high level components.

A key feature of the systems for which Erlang was primarily created is fault-tolerance. Erlang implements fault-tolerance in a simple way. Links between two processes A and B can be set up so that process B is eventually notified of the termination of process A and vice versa (using the normal message-passing machinery). The default behavior of a process that is informed of the abnormal termination of a linked process is to terminate abnormally itself. Alternatively the linked process can specify that it wishes to receive a message with a notification that its linked process has terminated. This process linking feature can be used to build hierarchical process structures where some processes are supervising other processes, and can take corrective action (e.g., restarting them) if they terminate abnormally. In order to create such fault-handling structures, Erlang/OTP provides the supervisor behavior.

Another key feature of Erlang systems, which is particularly useful for 24/7 systems, is the mechanism for hot code replacement. In short it is possible to phase out old code and replace it with new code, having both old and new code

running simultaneously. This feature enables bugs to be corrected and features to be added without stopping the system.

In summary, the Erlang/OTP programming environment is a comparatively rich programming environment for programming systems composed of (possibly) distributed processes that communicate by message passing. Fault tolerance is implemented by means of failure detectors (the linking mechanism), a standard mechanism in the distributed algorithms community. Moreover there is a process fairness notion, something which often makes it unnecessary to explicitly specify fairness in correctness properties. Moreover the language provides explicit control of distribution, and a clean model of distribution semantics. For distributed processes (processes executing on separate nodes) the communication guarantees are far weaker than for processes co-existing on the same processor node. This gives, in a clean way, considerable power with regards to checking a program under different environmental constraints (simply changing the mapping of processes to nodes), but on the other hand there is a requirement on implementing the run-time system with different guarantees for inter-node and intra-node communication.

Multi-core programming The concurrency oriented nature [3] and the (almost) transparent distribution makes Erlang a really good candidate for writing efficient distributed software. With the latest version of the Erlang Run-time System [13] this is taken even further, as it includes built-in support for SMP (Symmetric Multi Processing). SMP is today supported by most modern operating systems and becomes more and more important with the introduction of dual/quad/... processors, multi-core systems and hyper-threading technology. The SMP support in Erlang is transparent since most problems occurring in multi-threaded programs are solved by the Erlang VM. The SMP version of the VM can have many process schedulers running inside each OS thread, the default is to have as many schedulers as there are processors (or processor cores) in the system. Since the SMP support is completely transparent we get 'for free' an efficient multi-core implementation if we have a correct distributed implementation. This shows another benefit of having a working model checker for distributed Erlang.

3 Semantics

Erlang is at the same time both a simple language, having at its core a fairly uncomplicated dynamically typed functional language with eager evaluation, and a fairly complicated one. The complexity is due to the addition of language layers proving support for concurrency (processes and message passing), and distribution (processing nodes that encapsulate processes) and fairly elaborate inter-process fault detection and fault handling mechanisms (via process links and process monitors).

The intuitive picture of the distributed semantics is rather simple, the guarantees given are simply: "*communication between a pair of processes is assumed*

to be ordered" as described by Armstrong [3]. The semantics of links and monitors are also fairly easy to get an intuitive understanding of. However, the full semantics for distributed Erlang is indeed complex. It consists of some rather long and technical transition rules. Especially the corner cases, such as using the link mechanism on a dead process, makes a presentation somewhat lengthy and less intuitive than one could wish. Nevertheless, our formal description of the semantics is layered in three layers in a very clear way.

- **Functional Semantics -** consists of the pure functional part of Erlang (function evaluation, pattern matching, etc). It is dynamically typed and fairly straight forward.
- **Process Semantics -** is above the functional semantics, and consists of process evaluation rules (sending and receiving messages and links, starting/terminating processes, and silent computation steps) as well as process communication rules (process interleaving and process communication). This is all for the single node case, that is all the involved processes are executing in the same run-time system.
- **Node Semantics -** is placed on top of the process semantics, and adds the concepts of nodes and full distribution to the semantics. Similarly to the process semantics it consists of node evaluation rules and node communication rules.

The functional semantics and the process semantics are described in detail in Fredlund [15] and the node semantics is introduced in Claessen and Svensson [11]. The layering described here is, as we see later, clearly mirrored in the implementation of the model checker. Since it is not feasible to cover all aspects of the semantics in this paper, we just highlight a few important details. With the following example we show the importance of having the node semantics layer and that our intuitive understanding of the semantics is not sufficient in all cases.

3.1 World Hello?

Consider the small Erlang program in Fig. 1. When we run the function `world-hello()` it will spawn A, which in turn results in two processes being spawned (B and C). Thereafter A will first send the message `hello` directly to C and then send the message `world` to B. Process B is very simple, once it receives a message, it will forward it to process C. Process C just receives two messages, and prints the result. (`?MODULE` is a built-in macro which is replaced by the name of the current module by the compiler, $\texttt{?NODE}_i$ are ordinary macros defined elsewhere.)

The interesting aspect of this program is that the result of running the program depends on the distributed environment! If the program is running on a single node (that is `?NODE1 = ?NODE2 = ?NODE3`), the result is always: `hello world`. However if the program is running in a distributed environment (that is `?NODE1` $\neq$ `?NODE2` $\neq$ `?NODE3`), the result could be either `hello world` or `world hello`. The reason for this is that there are different communication guarantees at the distributed level. In short; in a single run-time system message delivery

```
worldhello() ->
    spawn(?NODE1,?MODULE,procA,[]).

procA() ->
    PidC = spawn(?NODE3,?MODULE,procC,[]),
    PidB = spawn(?NODE2,?MODULE,procB,[PidC]),
    PidC ! hello,
    PidB ! world.

procB(PidC) ->
    receive world -> PidC ! world end.

procC() ->
    receive X -> ok end,
    receive Y -> ok end,
    io:format("~p ~p\n",[X,Y]).
```

Figure 1: 'World Hello'-program

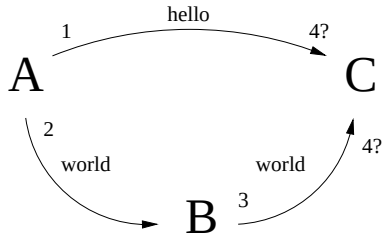


Figure 2: Possible message sequences

is instantaneous (that is the message is immediately put in the receivers in-box), while in a distributed system the only guarantee is that messages between a pair of processes are ordered. The possible message sequences in the distributed case is shown in Fig. 2.

The `etomcrl` tool for example (and the same goes for many other Erlang verification efforts) does not have a notion of nodes at all, and therefore this aspect cannot be checked. It is clear that this is a problem, since the difference in communication guarantees is a definite source of errors in Erlang systems (see for example Arts et al. [9]). It was therefore a strong requirement on McErlang that it should handle the node semantics. In fact, it is fair to say that the major part of the implementation effort of the model checker has been devoted to an accurate treatment of the often surprisingly complex semantics of the node semantics part of the run-time system.

3.2 Semantics implemented in McErlang

The McErlang tool has a full implementation of the distribution part of Erlang (i.e., explicit programmatic mapping of processes to explicit nodes), and thus provides the possibility to verify code based on either the assumption that all process are local (on the same node), or remote (all processes reside on different nodes), or a mix of the disciplines. Thus it is possible to verify a program under quite weak communication guarantees and be sure that later processes can be freely mapped on distributed nodes. However, the drawback of the distributed semantics is that it greatly increases the state space of the verified programs; essentially the distributed semantics non-deterministically delays the delivery of messages to a receiving process.

4 Structure of the Implementation

The model checker implementation is parametric, using the Erlang/OTP style of behaviors to specify particular component behaviors that provide services to the model checking algorithm.

The basic task of the model checker is of course to check a program against a correctness property, a *monitor module*, that implements the correctness property to check.

Except specifying which program to check (a specific Erlang function), and which Erlang module that implements the correctness property, a user of the tool can also choose:

- the name of a *language* module providing an operational semantics,
- the particular *verification algorithm* to use, (e.g., a safety property checker, a liveness property checker or just testing – i.e., simulation of the program in conjunction with a correctness property),
- the name of a *state table* implementation, that records encountered program states (typically a hash table), and
- the name of an *abstraction module* that abstracts program states,

The modular composition of McErlang is illustrated in Fig. 3, and in the following sections we describe the functionality of these modules in turn.

4.1 Source Language

The language module should provide two functions implementing an operational semantics for the language: (i) `transitions` which given a state returns a list of all next actions executable by the program, and (ii) the function `commit` which given an action returns a concrete program state. The `transitions` function may not cause side effects outside the model checker environment (e.g., really writing out a file to the file system) whereas `commit` may (if used by the simulation algorithm). The language module most commonly used is clearly the one providing

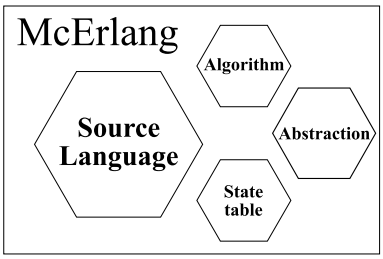


Figure 3: McErlang modular structure

an operational semantics for Erlang, however, we have also implemented an operational semantics for the WS-CDL web choreography language [30]. Although the effort is less mature than the Erlang model checker, it is interesting that the basic framework of the model checker can be reused in a different language setting [16]. As XML and XPath constitutes integral parts of the WS-CDL definition, having good support libraries available for these languages is very useful when representing their operational semantics. As Erlang has seen considerable industrial usage, the language already had good library support for working with XML based documents; we expect the same kind of advantages from using Erlang when providing model checkers for other target languages.

4.2 Correctness Properties

Correctness properties are encoded as automata programmed in Erlang. A safety monitor is a function which is checked in every reachable program state, and which returns an error if an invalid state is seen. A *Büchi monitor* (automaton) is a monitor that additionally may mark certain states as accepting. A program violates a Büchi monitor if a cycle can be found in the combined state space of the program and the monitor, which contains an accepting state. As is well known [29], linear temporal logic formulas can be automatically translated to Büchi automata.

The memory aspect of monitors is implemented by sending along the old monitor state as an argument to the Erlang function implementing the monitor. Concretely a monitor defines two callback functions: `init(`*`parameters`*`)` and `stateChange(`*`programState`*`,`*`monitorState`*`)`. The `init` function returns {`ok,`*`monState`*} where *`monState`* is the initial state of the monitor.

The `stateChange` function is called when the model checker encounters a new program state *`programState`* and the current monitor state is *`monitorState`*. If a safety monitor finds that the combination of program and current monitor state is acceptable, it should return a tuple {`ok,` *`newMonState`*} containing the new monitor state. If future states along this branch are uninteresting the monitor can return `skip` (e.g., to implement a search path depth limit), any other value signals a violation of the correctness property implemented by the monitor. A Büchi automatons should return a set of states, each state either accepting {`accepting,`*`state`*} or not {`nonaccepting,`*`state`*}. Normally we expect a "sound" `stateChange` function to be without side effects.

As an example, the code fragment in Fig. 4 implements a simple safety monitor that guards against program deadlocks: (a process is considered deadlocked if its execution state as recorded by the process data structure in the run-time system is `blocked`).

```
stateChange(State,MonState) ->
  case lists:any
       (fun (P) -> P#process.status =/= blocked end,
        State#state.processes) of
    true ->  {ok, MonState};
    false -> {deadlock, MonState}
  end.
```

Figure 4: Simple safety monitor

The syntax *variable#recordName.field* is used to access the field *field* of the record variable *variable*, of type *recordName*.

4.3 Algorithms

The McErlang tool currently offers two basic on–the–fly depth-first state traversal model checking algorithms, one to check safety properties and the other to check Büchi automatons (the liveness checking algorithm adapted from Holzmann et al. [21]). To give an intuition to the coding of these algorithms in Erlang, a schematic representation of the algorithm for safety property checking is depicted in Fig. 5 (we have abstracted out the parameter passing of modules implementing language (`Lang`), monitors (`Mon`), abstraction (`Abs`) and table implementation (`Tab`)).
To check an Erlang function call *m*`:`*f*`(`*p1*`,...,`*pn*`)`, given an initial monitor state *monState* and an empty state table *t*, and abstraction state *a*, the checking algorithm should be invoked with:

`check([[{mkProc(`*m*`,`*f*`,[`*p1*`,...,`*pn*`]),`*monState*`,`*t*`,`*a*`}]])`

where `mkProc` constructs a model checking process executing the function call argument.

As seen in the listing, model checking states are composed of a program state, a monitor state, a state table, and an abstraction state. Program states are checked against the monitor, and if accepted, are abstracted using an abstraction function provided by the module `Abs`. The abstracted states are checked against membership in the state table. If the program state is new, the set of next states is computed using the function `transitions`. Note that the particular choice of abstraction and table storage is abstracted out from the algorithm itself.

In addition there is a simple *simulator* available, which by default chooses the next program state randomly, but in addition has some debugging functionality, e.g., next states can be explicitly chosen, transitions can be single or multiple stepped, breakpoints can be set, and backtracking to previous states is supported. The simulator is also used to explore safety model checking counterexamples (traces).

```
check([]) -> ok;
check([[]|Earlier]) -> check(Earlier);

check([[State|Alts]|Earlier]) ->
  {ProgState,MonState,StateTab,AState} = State,

  % Check monitor
  {ok,NewMonState} =
    apply(Mon,stateChange,[ProgState,MonState]),

  % Abstract state
  {ok, {AbsState,NewAState}} =
    apply(Abs,abstractState,
          [{ProgState,NewMonState},AState]),

  % Check whether state already seen
  case apply(Tab,addState,[AbsState,StateTab]) of
    no ->
      check([Alts|Earlier]);

    {ok, NewStateTab} ->
      NewStates =
        [{S,NewMonState,NewStateTab,NewAState} ||
         S <-
          lists:map
          (fun (Action) -> apply(Lang,commit,Action),
           apply(Lang,transitions,[ProgState]))],
      check([NewStates,Alts|Earlier]])
  end
end.
```

Figure 5: Safety property checking algorithm

Fairness Constraints on Executions The Erlang language standard requires that process schedulers must be fair. The McErlang tool accordingly implements (weak) process fairness directly in its (liveness) model checking algorithm by omitting non-fair loops (i.e., ones that constantly bypass some enabled process) from the accepting runs.

4.4 Tables

A state table records pairs of program and monitor states encountered during model checking, to detect recurring states. The state table implementations used are normally imperative (e.g., updates to them are destructive) for performance reasons; however purely functional implementations of the tables are available.

```
-module(hashAbs).
-export([init/1, abstractState/2]).

init(Size) ->
  {ok,Size}.

abstractState(State,Size) ->
  {ok,{erlang:phash2(State,Size),Size}}.
```

Figure 6: Abstraction module for hashing

4.5 Abstractions

An abstraction abstracts a concrete program state into an abstract representation. It can be used to drastically reduce the checked state space of a program. The idea is inspired by the use of abstractions in Arts and Fredlund [5]. A typical abstraction used in model checking is to compute a hash value from the state, and to use the hash value as the abstract state when checking for membership in the state table. However, program specific abstraction functions can also be implemented. For example, an abstraction could transform an integer variable into a boolean value, signaling whether the integer is less than zero. Clearly, there is in general no guarantee that such an abstraction is safe, i.e., that it does not cause a program failure to escape undetected (false positive).

As a second example we have implemented the usual abstraction of collapsing a whole state to a single integer (through hashing), and using a bit array table module to implement the state table. Thus, in a modular fashion, we have obtained an implementation of Holzmann's bit-state hashing verification algorithm [20]. An implementation of a hashing abstraction thus becomes as simple as Fig. 6, where `erlang:phash2` is a built-in function which computes a hash value between 0..`Size` for its term argument. Note that is an unsafe abstraction, although as proven in practise in many verifications, also a highly useful one.

5 Executing Erlang Programs in McErlang

The model checking capability for Erlang programs is provided by executing Erlang programs directly in the existing Erlang run time system. This enables an easy and reasonably efficient handling of computations that act solely on data (the purely functional sub-part of Erlang). However, the existing Erlang run time system does not provide a method to capture the combined system state of a running program (check-pointing). This is unavoidable, since in general an Erlang computation could be distributed and so the combined state cannot be efficiently, or even reliably, collected.

For this reason we have implemented in Erlang a new run-time system for the concurrent and distributed part of the language, that implements easy access to the combined system state of an Erlang program. This run-time system simply simulates distribution and concurrency, all computations take place inside a single

real Erlang process. Structurally the new run-time system is layered on top of the old one, replacing only the process handling and the concurrency part of the old system. This layered structure also in many ways resemble the layered structure of the Erlang semantics in section 3.

Essentially a complete verification model consists of three parts: (1) an Erlang program containing the original program to be checked, (2) a re-usable implementation of the run-time system (also written in Erlang) and (3) a specification of the environmental constraints (e.g., which process/node failures and link failures occur). See section 6.2 for a concrete example of such environmental constraints. By separating the model cleanly into these three parts we can independently experiment with different assumptions/implementations. The workflow is illustrated in Fig. 7.

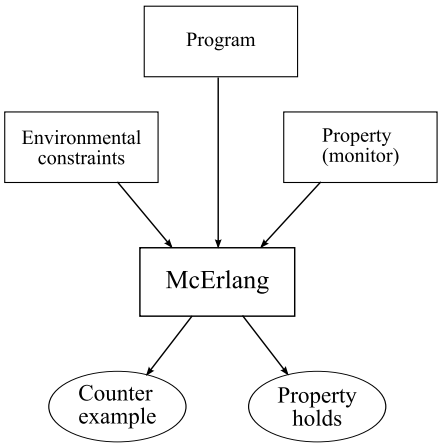


Figure 7: McErlang workflow

5.1 Run-time Organization

The state of the run-time system, e.g., recording process states, communication queues and so on (see section 5.3 for details), is stored in the imperative Erlang process dictionary; all (simulated) model checking processes run, interleaved, in a single Erlang process. All state updates and queries are thus implemented as accesses to the process dictionary. We have also experimented with a solution where the state of the run-time system is kept in a separate process, a solution more in the spirit of the Erlang design philosophy. Unfortunately, that solution severly impacts on the speed of model checking, slowing down a typical verification with a factor of three compared to the process dictionary solution. An obvious alternative would be to pass along the global state as a parameter everywhere in the verifier code. In e.g. Haskell cleaner as well as more efficient solutions are obviously possible.

5.2 Translation

A vital part of the model checker is a compiler that translates an Erlang program to be verified to a modified Erlang program that uses the new run-time system.

Actually we still use the old runtime system to execute even the translated functions (this is to avoid having to re-implement any part of the data handling in Erlang). However, calls to Erlang functions with side effects in the old runtime system have been replaced with calls to Erlang functions with side effects in the model checker instead.

The principal goal of the translation is to transform Erlang functions that use the `receive` construct[‡] so that instead of executing that construct, which would immediately hang the execution of the model checker as there would be no value to be received, the modified function instead returns a special return value. The return value indicates the desire to receive a message, and a continuation function coding the normal execution of the function after the reception of the message.

The translation takes a set of modules as input and returns a set of translated ones. The resulting Erlang modules can be compiled by the normal Erlang compiler (which is a requirement for using the model checker).

In case the compiled application makes use of OTP components (generic server, supervisor, etc...) the McErlang tool will include in the compilation the source code of tailored versions of these libraries, written in Erlang of course.

Replacing API calls Apart from transforming code that uses the `receive` construct, the translation does a very simple transformation of other API calls such as e.g. sending a value to a process.

As the Erlang language lacks a good reflection capability, the new run-time system is provided as a new application library `evOS`. For example, an application that used to send a message {`request,22`}[§] to a process with process identifier *`pid`* using the send construct *`pid`*`!{request,22}` should instead call the library function `evOS:send(`*`pid`*`,{request,22})`. The functions that implement the new API calls are implemented in Erlang itself and operate directly on the global system state (nodes, ether, processes, links, and a register map as discussed in section 5.3 below).

Handling Reception of Messages The mapping of calls to Erlang API functions to the new run-time system works for all Erlang constructs except the `receive` statement which is used by a process to retrieve a value from its mailbox (or process queue), as the `receive` call suspends until a matching value is available.

Instances of `receive` statements in the Erlang code to be model checked are instead replaced with code that returns a tuple like: {`recv,{`*`module, fun, context`*}, where the contained inner tuple {*`module,fun,context`*} identifies a function that implements the logic of the particular receive statement.

[‡] `receive` is a process construct to retrieve a value sent to the invoking process.

[§] A tuple containing a literal symbol `request` and the number 22. In Erlang variables begin with a capital letter and atoms (literals) with a lowercase letter.

When an invoked Erlang function, in an Erlang process, returns such a `recv` tuple the new run-time system recognizes the special return value and marks the process as `blocked`, and then checks whether there is any receivable value in the process mailbox (in which case the process status is upgraded to `receivable`). In any case, the run-time system can schedule another enabled process.

The transformation of an Erlang program containing a `receive` statement into one returning a `recv` expression is explained by the small example in Fig. 8 and Fig.9.

```
server(State) ->
  receive
    {new_state, NewState, Pid} ->
       Pid!{reply,State},
       server(NewState)
  end.
```

Figure 8: Receive statement – before translation

The code fragment in Fig. 8 defines a function `server` which guards some private state. The state can be changed by sending a call message to the server process, containing a process identifier and a new state. The server replies with the old state. The translation of the server function is shown in Fig. 9. In the transformed code, a call to `server(`*state*`)` will immediately return a tuple {`recv,{?MODULE,f_0,[`*state*`]}`} which is a special form recognizes by the model checker.

In general the function referenced in `recv` should accept two parameters, a message in the queue to be tested whether it is receivable, and a list of variables needed in the evaluation of `receive`. If the message is receivable, the function should return a tuple with a new anonymous function; if not `false` should be returned. The anonymous function receives the same parameters as the original function, and contains the body of the `receive` clause. The separation of receive into two functions serves to separate the testing whether a message is receivable from the actual retrieval of the message from the queue (as the process could continue by performing some side effect).

```
server(State) -> {recv, {?MODULE, f_0, [State]}}.

f_0({new_state, NewState, Pid}, [State]) ->
  {true,
   fun ({new_state, NewState, Pid}, [State]) ->
      evOS:send(Pid,{reply,State}), server(NewState)
   end};
f_0(_, _) -> false.
```

Figure 9: Receive statement – after translation

Handling a non tail-recursive `receive` The translation of the `receive` construct sketched above is correct only when it occurs in a tail-recursive position. For the general case, what is essentially a run-time stack is used instead.

The run-time stack is implemented using another special return value: {`letexp`,{*expr*,{*module*,*f*,*parameters*}}}, which is used in a situation where a receive statement occurs in an expression context (i.e. not in a tail-recursive position). Consider for example the recursive function `server` in Fig. 10 which repeatedly calls a function `doRequest` which in turn contains a receive statement.

```
server(State) ->
  {ok, NewState} = doRequest(State),
  server(NewState).
```

Figure 10: Non tail-recursive receive – before translation

The example in Fig. 10 is translated into a `letexp` return value as seen in Fig. 11. The function referenced in the `letexp` special expression is called when the inner function has returned a value, and receives as arguments the returned value as first argument and as second argument a list of variables necessary in the continued computation. In general all non tail recursive calls to functions that contain a `receive` in their body will have to be similarly guarded using a `letexp`. We use a global analysis over the set of input modules to the translator for computing the transitive closure of which functions may execute a `receive` statement.

```
server(State) ->
  {letexp, {doRequest(State), {?MODULE, f_1, []}}}.

f_1({ok,NewState}, []) ->
  server(NewState).
```

Figure 11: Non tail-recursive receive – after translation

The translation is somewhat complicated by the need to support the Erlang "feature" of permitting variable bindings to migrate out of their scope. The Erlang example in Fig. 12, which compiles without warning and does not cause run-time errors, illustrates the translation difficulty (`Logger` is assumed to be bound to a process identifier). Note that the variables `Msg` and `NewV` are bound in different branches of the `receive` construct, but may still be used outside of it.

Non-determinism in Erlang Another special return value is {`choice`,[{*module*, *fun*,*context*},...]} which introduces explicit non-determinism in Erlang; the model checker will non-deterministically select the continuation function from the list of function alternatives. This construct is

```
pingOrpong(Logger) ->
  receive
    {ping,V,Sender} ->
      Sender!{Msg=pong,NewV=V+1,self()};
    {pong,V,Sender} ->
      Sender!{Msg=ping,NewV=V+1,self()}
  end,
  Logger!{Msg,NewV},
  pingOrPong(Logger).
```

Figure 12: Migrating variable bindings

needed to use Erlang as a specification language rather as a programming one. As an example, suppose that we have implementing a drink machine in Erlang, offering either coffee or tea. Using the choice construct it is easy to model a machine user that non-deterministically selects either coffee or tea, and to verify that the program works correctly regardless what drink the user chooses (the model checker automatically explores both possibilities).

Finally {pause,{*module*,*fun*,*context*}} is short hand for a choice with a single continuation function; it is used to facilitate detection of interesting states in correctness properties.

5.3 Data Structures in the Run-time System

An Erlang state in our run-time system is a hierarchical structure and mimics to a large extent the organization of the real run-time system (and the structure of the layered Erlang semantics!) for Erlang, except, of course the state is physically centralized.

The top level of the hierarchical structure is composed of a tuple

$$\langle nodes, ether \rangle,$$

combining a data structure containing the nodes of the running system and an *ether* data structure containing messages in transit between nodes. Each message is identified by the following tuple:

$$\langle receivingNode, sendingNode, messageContent \rangle.$$

The ether data structure essentially has a separate queue of messages, sorted by sending time, for each pair of sending and receiving nodes. This is needed since the language guarantees that communication between any two nodes is FIFO-like, i.e., messages are delivered in order, if they are delivered at all. The *messageContent* contains the message itself (e.g., a normal message sent between two processes or a run-time event such as e.g. a notification of a process termination).

A node tuple

$$\langle name, processes, registered, monitors, node\_monitors, links \rangle,$$

is on the second hierarchical level. The *processes* field contains the processes executing on the node, *registered* implements the Erlang name server which maps (on a node basis) pids to symbolic names. The fields *monitors*, *node_monitors* and *links* is used in the three different process linking mechanisms available in Erlang.

Each process is a tuple

$$\langle status, expr, pid, queue, dict, flags \rangle.$$

The field *status* records the execution status of the process, e.g., whether it is blocked waiting on incoming messages, ready to run, or ready to receive an existing message. The *expr* field describes the next piece of code to execute, concretely a named user-defined Erlang function and a set of actual parameters to invoke the function with. The *pid* field is the system-wide unique process identifier of the process, *queue* contains the messages sent to the process that are available for reading (inter-node messages migrate from the *ether* data structure to the *queue* whereas intra-node messages are directly put in the *queue* data structure, mimicking the different communication guarantees provided by the run-time system for inter-node compared to intra-node communication). Finally *dict* contains a process dictionary (the equivalent of imperative variables in Erlang), and *flags* describes the setting of various process options.

Although the exact manner in which states are physically stored or represented (e.g., on the stack of 'choice points' and in the table of states previously seen) during a model checking is fully configurable, the normal exact representation of a state ensures that states are *normalized*, i.e., nodes are sorted in some order, as are processes within an nodes, as are links (pairs of processes identifiers in a node) and so on, to ensure a rapid check for state equality.

5.4 Model Checker Semantics

The tool implements a major part of the core `erlang` module in the Erlang/OTP distribution omitting mainly functions to inspect the run-time system itself, to obtain process status, timing functions, and ports (which are used to interface with foreign, i.e. non–Erlang, code). In total we provide around 40 such API functions, the implementation of which constitutes a significant portion of the lines of code of the model checker.

The operational semantics implemented by McErlang comprise an interleaving transition relation between Erlang states whose actions are decorated by sequences of actions (i.e., a big-step operational semantics). States are comprised by stable systems (e.g., where all processes are waiting in receive statements or have just spawned) and transitions are caused by invoking a single enabled process to run which may cause many side effects until it again becomes stable (waiting in a receive statement).

The use of a big–step semantics means that some errors will go undetected which would be caught using a smaller-step semantics. For the typically large scale systems that we are interested in verifying with McErlang there is a trade-off here. One option is to have a very detailed execution model with all the possibility

non-determinism inherent in the programming language.[¶] This quickly leads to enormous state spaces with the result that only a very tiny part of such state spaces can be explored by a model checker. On the other hand, we can reduce the non-determinism in the specification language by slightly changing its semantics. The result is smaller state spaces, which we can verify a bigger part of, but there are possibly states that we can never check because they will never be generated by the model checker. In future work we aim to implement a more finely-grained semantics for intra-node Erlang to explore this issue in further detail.

Interestingly it turns out that we can recover a more finely-grained semantics in case each process communicates only with other remote processes (located on other nodes). Then a send, as well as any other side effect, will be arbitrarily delayed (since the node *ether* data-structure is used, which essentially have separates queues for all pairs of communicating processes, see section 5.3 for details) compared to side effects caused by other processes, and so all interleavings of side effects are recovered.

5.5 Run-time Environment Modeling

Probably the most challenging part of developing a model checker for Erlang is to accurately model the environmental constraints put on a running Erlang program. For example: constraints on scheduling Erlang processes, the semantic impact of mapping processes onto remote processing nodes, the basic communication guarantees of Erlang, and on the frequency of failures in a running system.

Moreover the Erlang API has quite a few functions with side effects, whose actions cannot be understood as simply as sequences of lower-level primitives (send and receive) but are first-class citizens in any operational semantics.

As an example we consider below the implementation, which is a form of operational semantics, of the `erlang` API function `exit/2`. In Erlang, `exit(Pid, Reason)` is used to send a termination signal to the process referenced by `Pid`, which may be terminated as a result. The implementation has to handle the rather subtle interplay between fault-handling mechanisms (linking, monitors) and take into account process locality (on the same node, or not), etc. Moreover, its behaviour is very different depending on whether the process to terminate resides on the same node as the process executing the call or not.

Although the function may seem complicated, it is an intrinsic part of the Erlang language, which is used by programmers all the time (as invoked in through higher-level functions), and we have no choose but to model it faithfully if we wish to verify realistic Erlang software.

[¶]As an extreme case, Erlang, for instance, does not fix the order of evaluation of arguments to functions, so a totally faithful semantics would generate all such orderings. As Erlang programmers can happily write code that cause side effects in the evaluation of function call arguments, generating all such orderings may be highly important in model checking. However, the number of extra states could be huge, although part of the overhead could be eliminated through use of intelligent reductions. In practise, however, the only available Erlang language implementation *does* fix the order of argument evaluation, and in our opinion this is very unlikely to ever change in the future of Erlang.

Implementation sketch:

1. First the arguments are checked; if `Pid` is not a process identifier an exception is raised.
2. The code then checks if `Pid` is a local pid (i.e., the corresponding process resides on the same node as the process which executes the `exit/2` call. If the process is remote, a signal (a message) is sent to the node on which the process resides containing a request to issue an `exit/2` call, and the function returns.
3. If it is a local process, the process flags are retrieved. The process traps exit messages if the flag `trap_exit` is set. If `trap_exit` is set, and the `Reason` argument is not `kill`, a message, `{'EXIT',self(),Reason}`, is put into its mailbox (where `self()` evaluates to the pid of the process that called `exit/2`), and the function returns.
4. If the process is local, and it is not trapping exits, and the `Reason` argument is `normal`, the process is not terminated (and no message is put in its message queue), and the function returns.
5. Otherwise (the process is local, the reason is `kill`, or...) the process is terminated, i.e., it is removed from the process table.
6. Moreover any registered names for the process are removed (by modifying the *registered* element in the *node*).
7. And any monitors the now terminated process has set up are removed (all nodes are searched for such monitors), and messages concerning terminated processes due to such monitorings are removed (from the *ether* element).
8. Then every process that has requested to monitor the terminated process (information present in the *monitor* field of the *node* structure) are sent a message informing them of the termination of the process they monitored, and the reason for termination.
9. Then all the links mentioning the terminated process are examined (recorded in the *links* field of the *node* structure). If a link mentions a remote process, then the remote process is sent a signal (message) informing it that one of its linked process has terminated. If the process is local, the linked process is itself a candidate to terminate immediately, and execution contains for the linked process with roughly step 3 above.

As is indicated in the last step, in Erlang the termination of a process can, through the link concept, cause the termination of more processes, and so on, in a chain reaction. Although at first counter-intuitive, the idea is to use this behavior of the linking mechanism to write fault tolerant code. Essentially some processes are designated as supervisor processes, which are responsible for starting processes, and handling their termination by optionally restarting them. Such supervisor processes set the `trap_exit` flag to have termination message delivered

to their message queues. Their clients on the other hand generally do not set the `trap_exit` flag, since they do not contain programming logic to handle faults.

Many Erlang programs are written to be fault-tolerant, using the linking or monitoring mechanism, and although using ready-made components[‖] make the task easier, programming fault tolerant applications is still *hard*, and being able to check code under adverse run-time conditions using a tool such as our model checker is a significant help.

Ensuring Finite Models Clearly the efficacy of the model checking algorithm depends crucially on whether the checked Erlang program is finite state or not. However, note that for checking non-compliance this is not always necessary. For instance, we can easily code a monitor that raises an alarm whenever a process mailbox contains more than, say, N messages. Similarly, an abstraction (see the discussion in section 4.5) could simply cut the mailbox when it has grown too large.

Still, in model checking Erlang there are at least two sources of trivially infinite models that we need to avoid: the assigning of process identifiers to new processes, and the use of unique references to uniquely identify (generic server) calls. We solve both problems by consistently choosing the least *fresh* process identifier (or communication tag) absent from both the current program state and the correctness monitor.

6 Evaluation

To evaluate the use of McErlang we have used it on several non-trivial examples, ranging from a resource locker to a Video–on–demand server. Here we focus on two examples, first a simplified resource manager (or locker) originally implemented and verified by Arts et al. [8]. Their locker is based on a real implementation in the control software of the AXD 301 ATM switch developed by Ericsson. The second example is an implementation of a leader election algorithm. The implementation is (loosely) inspired by an algorithm presented by Singh [25]. Also this example originates from the AXD 301 ATM switch, but the particular implementation we studied here (and which have been studied before by Arts et al. [9]) is an open source version written by Wiger [31].

The two examples aims to show different aspects of McErlang, the locker example is comparing McErlang with `etomcrl` and does not use the distributed features of McErlang. On the other hand, the leader election example is distributed (and fault-tolerant) and the example shows that it is possible to find errors in a distributed application with McErlang.

Other case studies realized using McErlang include the verification of an implementation of the Chord peer–to–peer protocol [26], another implementation of a leader election algorithm namely Stoller's leader election algorithm [27], and of the above mentioned Video–on–demand server [17].

[‖] Such as, for example, the OTP supervisor pattern and the OTP generic server that are prepared to handle errors.

6.1 Resource manager

The locker is responsible for a number of resources, to which it can give clients exclusive or shared access, and which can survive client failures. To compare performance with the `etomcrl` tool we here focus on checking a single property**: is the locker safe with regards to mutual exclusion? That is, if a client requests exclusive access to a resource, and is granted access, then no other client will access the resource.

The source code of the example is split into four Erlang modules (files): (1) a module implementing a (parametric) client repeatedly accessing the locker using the `gen_server` OTP client-server component, (2) the source of a fault–tolerant locker, (3) a module implementing a supervisor process for starting the clients (using the `supervisor` OTP component), and (4) a supervisor that starts both the server and the client supervisor. In total around 430 lines of Erlang code.

The mutual exclusion monitor is provided in a separate Erlang module (around 60 additional lines of code); it checks whether multiple clients think they have access to the same resource, and at least one client has exclusive access (a mutual exclusion failure). In the client source we make visible the property of having access to resource by introducing a state using the `pause` value: {`pause,{?MODULE, inUse,[`*`Resources`*`]}`} which documents the resources and lock types the client thinks it has acquired.

Results As a comparison with `etomcrl` we present some figures for the checking of the locker example in table 1 below. The configuration column indicates, in a schematic manner, the model checking scenario used. For instance aEaEaEaEaS is a configuration with four clients requesting exclusive access to the resource a, and one client requesting shared access. The timing column shows the time for generating the transition system (for `etomcrl`, via the instantiator tool) and both the time to generate the transition system and check the mutex property for McErlang. The states column represents the number of states in the generated models. Note that for McErlang we use a non-lossy hash-table to store the state table.

	`etomcrl`		McErlang	
configuration	time	states	time	states
aEaEaEaEaE	52s	34282	17s	52197
aEaEaEaEaS	36s	28014	17s	50805
aEaEaEaSaS	39s	30814	18s	56313
aEaEaSaSaS	1m 4s	51928	25s	75801
aEaSaSaSaS	2m 49s	135038	42s	130101
aSaSaSaSaS	9m 29s	466702	1m39s	284277

Table 1: Comparison of `etomcrl` and McErlang

**Since `etomcrl` in contrast with McErlang does not support checking fault tolerance we did not introduce failures in the checked model; this was done in a separate experiment.

The table shows that in less complex scenarios, `etomcrl` creates smaller state spaces than McErlang. However, in complex scenarios (a scenario with more sharing is more complex, since many processes can request and succeed in getting a sharing lock on a resource at the same time) the difference in number of states evens out. The tool experiments were performed on a HP xw6400 workstation with four Intel Xeon CPUS each running at 1.60GHz (although neither tool made us of more than one CPU) and with 2 GB of memory, running Ubuntu 7.04.

It is hard to draw firm conclusions from the performance figures, although it is a promising sign that the time needed to generate the transition system using McErlang is competitive with the instantiator tool [32], as the instantiator is written in C and can be expected to be heavily optimized by now.[††]

6.2 Leader election

The objective of the leader election algorithm is to elect a leader among a fixed set of participants. This may seem trivial at first, but in a distributed and fault tolerant setting there are many subtle things that makes it a hard problem (and a well studied problem [24, 12] as well). Each *node* has a single leader election process, and the processes communicate with messages and also uses monitors to detect failures of other processes. There are two basic properties for leader election:

- **Safety** – two processes can never be elected as leaders at the same time.
- **Liveness** – eventually a process must be elected as the leader (or there is an infinite sequence of processes dying and restarting).

Both can easily be expressed as LTL-formulas (and hence as Büchi automatons). Here we focus mainly on the safety property.

To illustrate the typical organization of a verification we provide some details regarding the concrete files involved. The source code of the example is split into three Erlang modules (files): (1) a module implementing the leader election algorithm, (2) an environment for the leader election algorithm, and (3) a module that contains the monitor for the safety property. The test scenario is schematically illustrated in Fig. 13.

The environment module consists of code that initiates a set of nodes and starts a leader election process on each node. The environment also spawn controller processes (one for each node) that are responsible for killing and restarting the local leader election process. The controller processes in turn are dictated by a central stimuli generator (located on a separate node). The central controller sends messages to the local controller processes, which then enforces the order from the central controller (i.e., either killing or restarting the leader election process). All communication between conrollers are normal Erlang communication and it is all part of the model checking experiment. The reason for this somewhat strange stimuli generation structure stems from earlier testing, where we used *tracing* in a way which worked best with this structure. However, this

[††]Version 2.17.13 of the μCRL toolset was used.

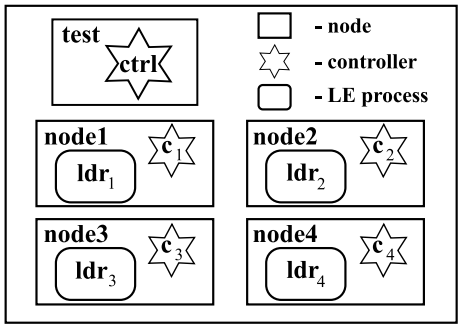


Figure 13: Leader election example organization

is a good example of one of the strengths of the everything–in–Erlang approach, where the code from testing can be re-used (almost as is) as the environment description in verification. Also, the flexibility of having the environment in a separate module (which consists of ordinary Erlang code) is that we could easily do a verification of only the start-up phase (or some other part of the state space, such as just killing the process with highest priority) by just changing the module with the stimuli code. Originally, the test code provides random stimuli, which is not very suitable for model checking. The randomness is removed in our example by setting the pseudo-random generator seed to a fixed value.

The monitor for the safety property is not very complicated, it only consists of a check if there are two leaders elected in the current system state. The property monitor is listed in Fig. 14. One thing that is clear from the listing in Fig. 14 is the need for a set of convenience functions for accessing the states and retrieving information from the state.

Results If the example is run in McErlang using the safety algorithm, and the `NotTwoLeaders` monitor the result is a counter example. The time it takes to reach a counter example is only a few seconds (depending on the seed choosen it can take longer or shorter time) on a fairly modest workstation. The size/length of the counter example includes around 50 transitions. The existence of a counter example is not surprising, since other studies of the same algorithm [9] have revealed errors. (The counter example described below is actually exactly the same as the one labeled 'The first serious bug' in that paper)

The counter example scenario is described in Fig. 15. The problem in the scenario is that some communication is slower than other. Since in the protocol only a majority of the involved processes needs to accept a candidate it is possible that an existing leader (B in the scenario) could be outnumbered by newly started and fast communicating processes (A and C in the scenario).

What is important to note is that the error found is only present in a distributed and fault tolerant semantics. That is, we could not have found this error using a model checker (or other verification tool) that does not support the distributed semantics of Erlang. We also have the possibility to search for the

```
-module(monNotTwoLeaders).

init(State) ->
  {ok,{safety,State}}.

stateChange(State,MonState,_) ->
  case notTwoLeaders(stRecords(allProcs(State))) of
    true ->  {ok,State};
    false -> {error,stRecords(allProcs(State))}
  end.

allProcs(State) ->
  lists:flatmap
    (fun (Node) -> Node#node.processes end, State#state.nodes).

stRecords([]) -> [];
stRecords([P|Rest]) ->
  case P#process.expr of
    {recv,{ev_gen_server2,_,{Rec,_}}} ->
      [Rec|stRecords(Rest)];
    _ ->
      stRecords(Rest)
  end.

isLeader({P,{_,State}}) ->
  Ldr = State#data.leader,
  P#process.pid == Ldr.

notTwoLeaders(States) ->
  length(lists:filter(fun isLeader/1,States)) < 2.
```

Figure 14: Safety property monitor – NotTwoLeaders

```
Three processes A,B,C (with priority A > B > C):
B is started
B: Send 'capture' to A,C and monitor A,C.
B: Receive 'Down' from A.
B: Receive 'Down' from C, broadcast 'elected'.
B is the leader
A is started
C is started
A: Send 'capture' to B,C and monitor B,C.
C: Receive 'capture' from A, Send 'accept' to A.
A: Receive 'accept' from C, broadcast 'elected'.
A is the leader
```

Figure 15: Counter example from leader election

shortest path leading to an error (again what is the shortest vary due to the introduced randomness). Having the shortest counter example is often desirable since it includes the least amount of unnecessary information. A search for the shortest path to an error is of course slower, sometimes several order of magnitudes slower. In one of our examples a search took about 30 minutes, and explored somewhere around 10 million states.

7 Discussion

In this section we want to discuss some alternative implementation aspects. As mentioned in the introduction we made some experiments with a prototype implementation of the distributed Erlang semantics in Haskell. The prototype consisted of an Erlang parser and a layered run-time system with flexible control of path choice, etc. It supported all the distributed features of Erlang, but a lot of the more basic pure functional things were missing.

We asked ourselves if it would be possible to use such an implementation as the starting point for a model checker for Erlang as well. Much of the work with McErlang has gone into accurately modeling the node level semantics of Erlang. Starting instead with an implementation of the distributed semantics that task would be much simpler. We also think that a lot of the modular structure of McErlang could be the same in a Haskell implementation.

We have identified some advantages with a Haskell approach as well as some drawbacks. One of the major drawbacks is that one looses the ability to re-use the existing evaluation mechanisms for the purely functional part. This means that every lower-level built-in pure function and data structure has to be dealt with in the implementation. To implement this is perhaps not a very complicated task, however we deemed it as far too time-consuming for a research project. On the other hand, by having full control of the whole run-time system we could omit the Erlang–to–Erlang compilation phase discussed in section 5.2. It would also be trivial to switch from a *big-step* semantics to a *small-step* semantics since we could easily turn other syntactic constructions into choice points. A final drawback is of course also that we miss the "all-in-Erlang" aspect, since we involve Haskell. This could be a hinder for an experienced Erlang programmer with limited Haskell knowledge.

Our conclusion is that it is certainly possible to implement the same type of model checker in Haskell. However, it seems to be a lot more time-consuming, and it is not obvious that the end result would be any better than McErlang.

8 Related Work

Software model checking is a very active research field, which means that there exist an overwhelming amount of related works. We try to mention the most important and the ones which have provided inspiration for McErlang.

For Erlang the `etomcrl` toolset [7] already provides a model checking capability. Although it is more restricted, covering a smaller subset of Erlang, for

instance lacking the concept of distribution and fault tolerance (i.e. nodes, processes, links, monitors, ...). Other verification tools for Erlang include Huch's abstract interpretation model checker [23] which uses abstract interpretations to reduce the size of the state space. We also have the "*Verification of Erlang Programs*"-project [18] which uses theorem proving technology. Further there is the interesting QuickCheck tool for Erlang by Arts and Hughes [6], which however is more of a testing tool than a verification tool as it cannot detect recurring states.

The work on tracing for Erlang, in particular the approaches that have used *abstractions* to handle the size of the traces, by Arts and Fredlund [5] and by Arts et al. [9] was also a source of inspiration for the abstraction part of the McErlang implementation.

A lot of the inspiration for this work naturally comes from the work on the SPIN tool by Holzmann [20] and the CADP toolset [14], as they both constitute very capable language based platforms for the verification of software, and for testing new verification algorithms.

The VeriSoft tool by Godefroid [19] is one of the earlier examples of providing a verification functionality to a real, complex, programming language (such as C or C++) instead of a simpler specification language. Another successful example of such a verification project is the Modex tool [22] which is closely connected to SPIN. A recent work on the verification of complex concurrent program code is the work on model checking file system implementations by Yang et al. [33]. Another recent work is the Zing model checker by Andrews et al. [1] which aims at checking concurrent systems.

9 Conclusion and Future Work

As we have seen, adopting an "everything-in-Erlang" approach to model checking has certain advantages. It is easy to provide a rich specification language, and to use the same language for formulating correctness properties as for programming is convenient. Moreover much of the basic execution machinery can be reused (e.g., McErlang uses the normal Erlang run-time system extensively). The result is a model checker for Erlang, which supports all aspects of distribution and fault tolerance. This is especially important since distributed and fault-tolerant implementations are known to be error prone and hard to test and debug. It is our hope that McErlang is also simple enough to use, such that it can be used by the ordinary Erlang programmer.

With two examples we have compared McErlang with the existing `etomcrl` tool set and also showed that it is indeed possible to find errors in a distributed program using McErlang. The performance of McErlang looks promising, and the trade-off between expressive power and efficiency seems positive. However, more case studies are needed before we can be certain about the capacity of McErlang.

Another good property of McErlang, is the clearly separated input. We can easily experiment with different environment constraints for a program under test. This is particularly useful if one is only interested in part of the complete state

space, since the search space could easily be altered by changing the environment constraints as we saw in the leader election example in section 6.2.

We have also experimented with an alternative implementation approach using Haskell. There we concluded that although it is a possible alternative it is far from obvious that the result would be better than McErlang.

During the development of the McErlang tool we also realized that a (dynamically typed) functional language offers several advantages over traditional languages like C as a general framework for implementing formal verification tools (e.g., quick prototyping, clean higher-order functions, separating functionality cleanly into modules, seamless composition of modules, and so on). Thus we have started experimenting with the use of the McErlang tool as a general framework for building model checkers for various target languages. Essentially this involves provides an executable operational semantics for the target language in question, together with the glue necessary (state parsers and unparsers, and so on). As a small experiment we implemented a simple interpreter and model checker for the WS-CDL web choreography language [30].

Future work The tool is far from finished, there are many things that we want to investigate further, the following list indicates some of these areas:

- We would like to experiment with partial-order verification algorithms for the model checker. Clearly such reductions are normally quite language specific, and it will be instructive to see whether we can express their enabling conditions cleanly in Erlang. Moreover we can hope to benefit from the fact that standard components are heavily used in Erlang, which should results in more regular communication exchanges, i.e., which are more amenable to partial order reductions.

- To use McErlang on a larger body of programs we need to support a slightly richer Erlang fragment (e.g. the `port` construct for communicating with the external world). In particular it would be interesting to have a normal Erlang node communicate with nodes in our "modeled" Erlang environment.

- We should provide the option of changing the Erlang semantics implemented in the tool to re-schedule processes not only when a receive statement is encountered, but to do so for every side-effect inducing operation (e.g. message sends). This will result in a small-step semantics option that may detect new program bugs.

- Since many aspects of Erlang (asynchronous message passing, rich error detection mechanisms and process fairness) closely match standard implementation environments for distributed algorithms. Therefore, it seems reasonable to think that McErlang can be really useful also for verification of general distributed algorithms. The *leader election algorithm* example, presented in section 6, could be seen as one example of such an algorithm.

- We would like to develop a library of useful state abstractors for Erlang to enable this part of the tool to see wider use.

Acknowledgement

Thanks are due to Clara Benac Earle, Juan José Sánchez Penas, Koen Claessen and Thomas Arts.

References

[1] T. Andrews, S. Qadeer, S.K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Lecture Notes in Computer Science*, volume Vol. 3114, p. 484 – 487, Jan 2004.

[2] J. Armstrong. *Programming Erlang – Software for a Concurrent World*. The Pragmatic Programmers, http://books.pragprog.com/titles/jaerlang, 2007.

[3] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, December 2003.

[4] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

[5] T. Arts and L. Fredlund. Trace analysis of Erlang programs. *SIGPLAN Not.*, 37(12), 2002. ISSN 0362-1340.

[6] T. Arts and J. Hughes. QuickCheck for Erlang. In *Proceedings of the 2003 Erlang User Conference (EUC)*, 2003.

[7] T. Arts, C. Benac Earle, and J. J. Sánchez Penas. Translating Erlang to mucrl. In *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD2004)*. IEEE Computer Society Press, June 2004.

[8] T. Arts, C. Benac Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2–3):205–220, March 2004.

[9] T. Arts, K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. *Lecture Notes in Computer Science*, 3395:140 – 154, January 2005.

[10] S. Blau and J. Rooth. AXD 301 - a new generation ATM switching system. *Ericsson Review*, 1:10–17, 1998.

[11] K. Claessen and H. Svensson. A semantics for distributed Erlang. In *Proceedings of the ACM SIPGLAN 2005 Erlang Workshop*, 2005.

[12] S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Trans. Parallel Distrib. Syst.*, 8(4):424–440, 1997. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/71.588622.

[13] Erlang 5.5/OTP R11B. The Erlang/OTP Team. URL `http://www.erlang.org/doc/doc-5.5/doc/highlights.html`.

[14] J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, p. 437–440. Springer, 1996.

[15] L. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

[16] L. Fredlund. Implementing WS-CDL. In *Proceedings of the second Spanish workshop on Web Technologies (JSWEB 2006)*. Universidade de Santiago de Compostela, November 2006.

[17] L. Fredlund and J.J. Sánchez Penas. Model checking a VoD server using McErlang. In *In proceedings of the 2007 Eurocast conference*, Feb 2007.

[18] L. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):405 – 420, Aug 2003.

[19] P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *Computer Aided Verification*, p. 476–479, 1997.

[20] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[21] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, p. 23–32. American Mathematical Society, 1996.

[22] G. J. Holzmann and M. H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. Softw. Eng.*, 28(4):364–377, 2002. ISSN 0098-5589.

[23] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, 1999.

[24] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[25] G. Singh. Leader election in the presence of link failures. In *IEEE Transactions on Parallel and Distributed Systems, Vol 7*. IEEE computer society, 1996.

[26] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, p. 149–160, 2001. URL `citeseer.ist.psu.edu/stoica01chord.html`.

[27] S. D. Stoller. Leader election in distributed systems with crash failures. Technical Report 481, Computer Science Dept., Indiana University, May 1997. Revised July 1997.

[28] S. Torstendahl. Open telecom platform. *Ericsson Review*, 1, 1997.

[29] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. p. 332–344, 1986.

[30] W3C. Web Services Choreography Description Language, Version 1.0 – W3C candidate recommendation 9 november 2005. Technical report, W3C, November 2005.

[31] U. Wiger. Fault tolerant leader election. URL `http://www.erlang.org/`.

[32] A.G. Wouters. Manual for the μCRL toolset. Technical report, CWI, Amsterdam, 2001.

[33] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Sixth Symposium on Operating Systems Design and Implementation*, p. 273–288. USENIX, 2004.

Paper 5

A Semi-Automatic Correctness Proof Procedure applied to Stoller's Leader Election Algorithm

This document was published as Technical Report no. 2008:7 at Computer Science and Engineering, Chalmers University of Technology, 2008.

A Semi-Automatic Correctness Proof Procedure applied to Stoller's Leader Election Algorithm

Hans Svensson

Technical Report no. 2008:7
Chalmers University of Technology, Göteborg, Sweden

hanssv@cs.chalmers.se

Abstract

In 1997, Stoller presented a leader election algorithm for a synchronous system with crash failures. The algorithm is an adaptation of Garcia-Molina's *Bully Algorithm* that uses failure detectors instead of explicit timeouts. Since the characteristics of the algorithm closely resemble the Bully Algorithm Stoller does not give a formal correctness proof. However, although the algorithms appear similar, there are non-trivial differences. The differences make it unclear if the original proof, by Garcia-Molina, actually carries over as indicated by Stoller. In this document we formalize the leader election algorithm using first-order logic, and prove its correctness with respect to the obvious safety property; it should not be possible to elect two different leaders at the same time.

1 Introduction

This report describe our efforts to prove the correctness of a fault-tolerant leader election protocol by Stoller [Sto97]. We have used *semi-interactive* theorem proving to formally verify safety properties of the protocol. Here, semi-interactive means that we have manually constructed invariants from which we have automatically generated proof obligations. The proof obligations are solved automated theorem provers. Most proof obligations can be proved automatically; however, a few of the generated proof obligations needed some manual (high-level) interaction to be provable.

In distributed systems, the task of leader election, where a fixed set of processes has to determine a special process, the *leader*, is both important and complex. Leader election algorithms are used for work co-ordination in distributed systems, also, a leader election algorithm is often the basis for a more intricate high-level algorithm. For leader election there is one basic safety property *"there should never be more than one leader"*, and one basic liveness property *"eventually there should be a leader"*. In a *fault-tolerant* setting, the participating processes can crash and restart at any point in time, making the problem significantly harder.

The leader election algorithm we have verified was introduced by Stoller in [Sto97], it is closely related to the classical leader election algorithm, *The Bully Algorithm*, by Garcia-Molina [GM82]. The verification of this Bully algorithm is part of an effort to verify an Erlang implementation of an (by us) adapted version of the Bully algorithm. The Erlang implementation was developed by us, after we discovered some rather subtle bugs in an earlier implementation (not based

on the Bully algorithm) using trace-based testing techniques [ACS05]. The new implementation is based on Stoller's algorithm [Sto97]; however, to fully meet the requirements and fit into the Erlang framework we had to make some non-trivial changes to the protocol.

From our previous experience, we knew that it is extremely hard to correctly implement this kind of algorithms. And although the implementation withstood thorough testing using the same techniques that uncovered the bugs in the previous version, we were not fully confident that the implementation was correct. One reason for our doubts was that Stoller never gives a formal correctness proof for the algorithm in his paper. Stoller's algorithm is a slightly adapted version of a classical leader election algorithm by Garcia-Molina, which in turn is only informally proved correct in the original paper [GM82]. Stoller claims that his modifications are so minor that there is no need to give a new proof: *"The proofs that the Bully$_{FD}$ Algorithm satisfies SLE1 and SLE2 are very similar to the proofs of Theorems A1 and A2 in [GM82] and are therefore omitted."*

Therefore before trying to formally verify the algorithm in the implementation, which is a bit more complicated than Stoller's original algorithm, we decided to verify Stoller's algorithm. We tried several different model checking methods (among others SPIN [Hol03] and our own model checker McErlang [FS07]). These model checkers, although very capable, could not handle the problem except for very small and unconvincing bounds on the number of processes, message queue sizes, etc. This is mostly due to the extremely high number of states, which are generated by the combination of fault-tolerance and asynchronous message passing.

We found a method, where we prove invariants of the system inductively, that worked well. We use first-order logic theorem provers to solve proof obligations generated by our model of the algorithm as an abstract transition system and the invariants. By using first-order logic we could prove properties about the system for any number of processes, unbounded message queues and an unbounded number of occurring faults. The first-order logic theorem provers we used were very capable, and the level of interaction was mostly at the level of stating the correct invariants. (However, some proof obligations also needed some manual interaction before being provable.)

The main problem of the approach is to come up with the right invariant(s), unfortunately this is a non-trivial task. Most of the time the invariant that one is trying to prove is not inductively provable. Not being inductively provable means that we have to strengthen the original set of invariants by adding new invariants; until we have an inductively provable set of invariants. In our case we started with the single invariant *"There should never be more than one leader"* and finished off with a set of 89 invariants!

Contributions – In this document we present a formalization and formal correctness proof of the Bully Algorithm as stated by Stoller [Sto97]. The proof methodology used should be generally applicable, and some effort has been made in that direction. The same style of reasoning should be applicable for other distributed and fault-tolerant algorithms.

Document organization – In Sect. 2 we give some background of the fault-tolerant leader election protocol. In Sect. 3 we describe the actual algorithm. The proof procedure is described in more detail in Sect. 4. Sect. 5 contains a detailed description of the model of the algorithm and the algorithm environment used in the proof. In Sect. 6 we explain the invariant that we want to prove. Some implementation effort was made in order to represent the algorithm and carry out the proof steps, this is briefly presented in Sect. 7. In Sect. 8 some numbers and examples from the proof procedure are presented together with a short summary of the automated theorem provers used in the proof. Sect. 9 contains a short summary. Finally, in the Appendix all the used invariants are presented together with a listing of the axioms and a description of the logic functions and predicates.

2 Background

In 1982 Garcia-Molina published a classic paper on 'Elections in a Distributed Computing System' [GM82]. In that paper Garcia-Molina describes and discusses failure environments and also presents two different leader election algorithms *the Bully Algorithm* and *the Invitation Algorithm*. The Invitation Algorithm is specified for the more complex situation where messages can get lost and participating processes may be arbitrarily slow; whereas the Bully Algorithm works in the less hostile situation where only crash failures are considered. In this document we focus on the Bully Algorithm, and for many modern distributed computing systems the required assumptions are actually fulfilled.

In 1997 Stoller presented a paper, 'Leader Election in Distributed Systems with Crash Failure' [Sto97]. In this paper Stoller gives a more general formulation of the Bully Algorithm, where *failure detectors* are used to make the algorithm more modular. Stoller also points at a minor flaw in Garcia-Molinas specification for the Invitation Algorithm. Stoller claims that the modifications made to the Bully Algorithm are minor. In fact they are so minor that no new proof is needed, and Stoller refers to the proof in Garcia-Molina's paper. However, the proof in Garcia-Molina's paper is only informal, although rather detailed, and thus it is not an entirely convincing statement by Stoller.

In this document we formalize the Bully Algorithm by Stoller, using first-order logic, and prove its correctness by the use of automated theorem provers. The formalization is somewhat colored by the fact that it is part of a verification effort for Erlang programs. Some parts of the model are actually not the most straightforward solution, but rather the most Erlang-ish. To give an example, we decided to implement the failure detection, which is a cleanly separated mechanism in Stoller's description, in terms of messages more akin to failure detection in Erlang.

3 Algorithm

3.1 General Description – The Bully Algorithm

The algorithm presented by Garcia-Molina was named the Bully Algorithm, because in the election process nodes with a high priority force nodes with a lower priority into accepting them as the leader. (Note, in the original paper the term *coordinator* was used instead of leader. In the original algorithm there was also a *reorganization* phase before normal operation was started upon electing a new leader.) The Bully Algorithm uses node identification numbers as priorities. Garcia-Molina assigns highest priority to high numbers while Stoller takes the Unix approach and assigns high priority to low numbers. Since we are mostly studying Stoller's algorithm, from now on we assume that the lowest node identification number means the highest priority. In the following we are using the terms node and process somewhat interchangeably, since there is never more than one leader election process alive at the same node. However, different incarnations of the leader election process have different process identifiers to avoid confusion in communication.

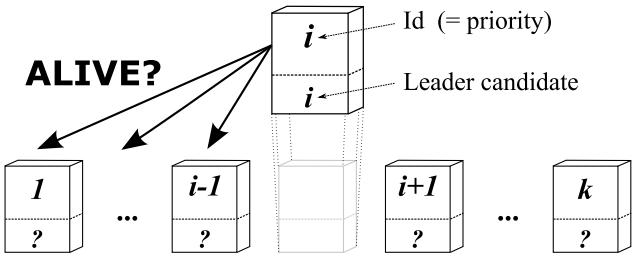
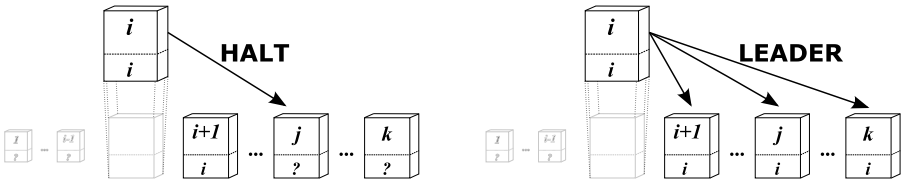


Figure 1: Election phase 1



(a) Part 1 (b) Part 2

Figure 2: Election phase 2

To describe the algorithm, we look at the node with identification number i. Assume that i starts an election. (There are several reason for starting a new election; the old leader died, the process just recovered or the last election was aborted since the candidate leader died.) The election protocol consists of two phases. The first phase includes communication with nodes that have a higher priority and the second phase consists of communication with nodes that have a lower priority. In the first phase, as shown in Fig. 1, node i tries to contact all

nodes with a higher priority. If any of them is alive, node i gives up its bid to become leader and waits for communication from that node. (If node i does not hear from this node in a while, it should again restart the election process.) If none of the nodes with a higher priority answers, then node i continues with the second phase of the algorithm. In the second phase, node i contacts all nodes with a lower priority (in order) to inform them that node i intends to become the leader. This is done in a two-step manner by first, as shown in Fig. 2(a), forcing all nodes with a lower priority into a state where they are ready to accept the new leader. Second, to actually become the leader, node i sends an appropriate message to all nodes with a lower priority, as illustrated in Fig. 2(b). (To get a better intuition on why this works one can read the informal introduction of the algorithm in Garcia-Molinas paper [GM82].) If i itself has the highest (or the lowest) priority, the first (or the second) phase is trivial.

3.2 Garcia-Molina's Bully Algorithm

In Garcia-Molina's presentation of the Bully Algorithm a RPC-like construction with an explicit timeout is used. A typical procedure call looks like:

CALL proc(i,`parameters`) ONTIMEOUT(t): *stmt*.

That is, detection of failure is done by an explicit timeout, and a node simply calls an empty procedure on another node to check if it is alive. The advantage of using this type of communication is that the caller is blocked until either the procedure is remotely executed or the call timeout, this asserts a high degree of sequentiality. Sequential actions simplifies algorithm design and reasoning about the algorithm. At the same time this is a drawback. One major point of distribution is to do things simultaneously, for example checking that all nodes with a higher priority are dead can be done efficiently in parallel.

3.3 Stoller's Bully Algorithm

Stoller's version of the Bully Algorithm is not very different from Garcia-Molina's. Stoller introduces modular *failure detectors* and the use of ordinary communication instead of RPC-calls makes his version less sequential. A failure detector is a module that detects and reports crashes. A typical failure detector can be *started* and *stopped*. Usually the start and stop actions are parametrized with the identifier of the process that should be monitored. When a process monitored by a failure detector crashes a notification of the crash is sent to the process initiating the failure detection. By using failure detectors it is possible to perform phase one of the election (checking that all nodes with a higher priority are dead) in parallel. In Stoller's Bully Algorithm a failure detector is initiated for each node with a higher priority, and the crash notifications are collected in the order they arrive. Each process participating in the election protocol has a few state variables, described in Tab. 1. More details of Stoller's Bully Algorithm are discussed in Sect. 5 where our model of the algorithm is described.

pid	The process identifier.
status	The election status, one of: *elec_1* – the first phase of the election, *elec_2* – the second phase of the election, *wait* – the status for a process that has given up its bid for the leader role, and *norm* – the status for all stable processes (both the leader and the ones recognizing the leader have status *norm* when the election is over).
ldr	The host identifier of the leader; this variable only has a valid content when a process has status *norm*.
elid	The election identifier; used by the process to keep track of who is its leader candidate.
down	The set of processes known to be dead.
acks	The set containing the processes from which an `Ack`-message has been received.
pendack	The counter used to keep track of where in the first part of the second phase the process is. (I.e., who is the process waiting for an `Ack`-message from.)

Table 1: State variables in Stoller's Bully Algorithm

4 Proof Procedure

The proof procedure used in the verification needs three components, which describe the verified system and the invariants:

- A predicate $Init$ describing the initial state,
- A predicate Inv describing the invariant,
- A predicate transformer $[Sys]$ that abstractly describes one transition of the system.

For the predicate transformers, we borrow notation also used in dynamic logic [BHS07] and the B-method [Abr96, Wor96]. For a program S and a post-condition Q, we write $[S]Q$ to be the weakest pre-condition for S that establishes Q as a post-condition. This in turn means that we can write

$$P \rightarrow [S]\, Q$$

which has the same meaning as the Hoare triple $\{P\}S\{Q\}$ [Hoa69].

The language we use to describe Sys is very simple. The three most important constructs are assignments, conditionals, and non-deterministic choice. The definition of predicate transformers we use is completely standard, and we will only briefly discuss the concepts here. For more details, the reader can consult [Abr96]. Here are the definitions of the predicate transformers for assignments,

conditionals, and non-deterministic choice, respectively.

$$\begin{array}{rcl} [x := e]\, P & = & P\{e/x\} \\ [\textbf{if}\, Q\, \textbf{then}\, S\, \textbf{else}\, T]\, P & = & (Q \rightarrow [S]P) \wedge (\neg Q \rightarrow [T]P) \\ [S \mid T]\, P & = & [S]P \wedge [T]P \end{array}$$

Establishing Inv as an invariant amounts to proving the following two statements:

$$\begin{array}{rcl} Init & \rightarrow & Inv \\ Inv & \rightarrow & [Sys]\, Inv \end{array}$$

In practice, Inv is really a conjunction of a number of smaller invariants:

$$\begin{array}{rcl} Init & \rightarrow & Inv_1 \wedge Inv_2 \ldots \wedge Inv_n \\ Inv_1 \wedge Inv_2 \ldots \wedge Inv_n & \rightarrow & [Sys]\, (Inv_1 \wedge Inv_2 \ldots \wedge Inv_n) \end{array}$$

The above two proof obligations are split up into several sub-obligations; for the initial states, we prove, for all i, several obligations of the form:

$$Init \rightarrow Inv_i$$

For the transitions, we prove, for all i, several obligations of the form:

$$\left(\bigwedge_{j \in P_i} Inv_j \right) \rightarrow [Sys]\, Inv_i$$

That is, for each invariant conjunct Inv_i, we have a subset of the invariants P_i that we use as a pre-condition for establishing Inv_i. Logically, we can use all invariants Inv_j as pre-condition, but in practice the resulting proof obligations would become too large to be manageable by the theorem provers we use. Also, it is good to "localize" dependencies, so that when the set of invariants changes, we only have to redo the proofs for the obligations that involves the invariants we changed.

To simplify the proof obligations as much as possible, we also use an aggressive case-splitting strategy. Thus each of the above proof obligations is proved in many small steps. The case-splitting is fully automatic and further discussed in Sect. 8.

4.1 Sanity Checks

The complete proof is very complex and consists of quite a few generated and hand-written first-order logic formulas. Therefore, there are many ways in which the reasoning could be compromised. For example by introducing a bad axiom which makes everything provable, or by having a too abstract model which accepts also incorrect algorithms. Therefore it is good to do as many sanity checks as possible, here we mention two strategies for doing sanity checks.

Breaking the Algorithm

By deliberately breaking the algorithm we should see that we no longer are able to prove all of the invariants. There are many ways in which the algorithm could be broken. However, we did not do any systematic experiments but rather did a few ad hoc changes to the algorithm which all resulted in one or more failing proof attempts. These experiments should for example reveal if the axioms are inconsistent.

Removing Invariants

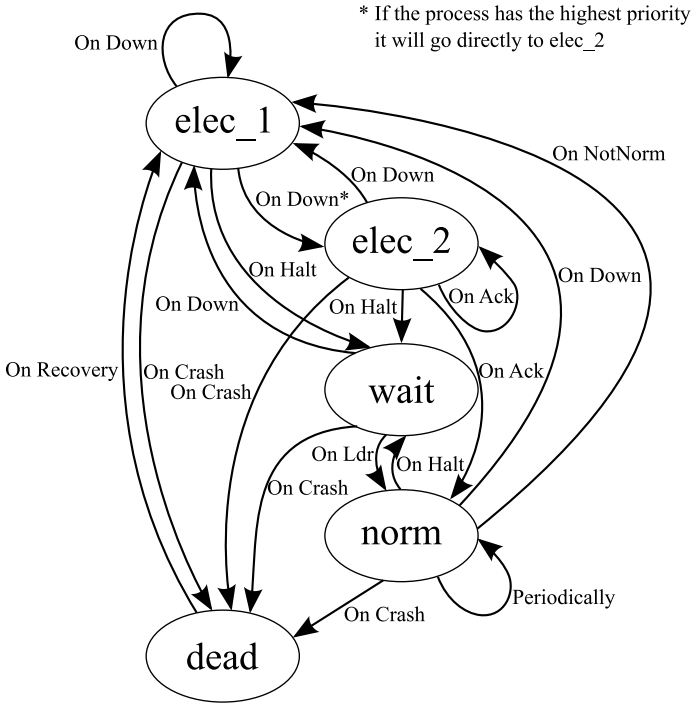
Another way to test the robustness of the proof is to deliberately remove needed sub-invariants from the set of invariants in the antecedent. Since these sets are not necessarily *minimal* we should not expect this test to render all problems unprovable.

5 Model

In our formalization of Stoller's version of the Bully Algorithm, we do not use a separate failure detection module as in Stoller's presentation. For simplicity, and more like in Erlang, we instead over-approximate the failure detection by actually informing every alive process about every failure via messages. The over-approximation could be a potential problem, but the algorithm is designed robustly with regard to getting too many failure reports. The algorithm code is shown in Fig. 4, 5 and 6. The algorithm is presented in a reactive way, the **On** `Msg` construction is what is executed when a process receives Msg. Also *Periodically, Crash, Recovery* means exactly that, i.e. the code snippet is executed periodically, on process crash and on recovery respectively. (Note that only processes that are alive execute code periodically. A crashed process can only execute the recovery code.) In Fig. 3 we show a simplified overview of the state space of the model. Note, in reality there is no explicit *dead*-state, although there is one in the figure to make it more readable.

Each reactive sub-part of the program corresponds to a predicate transformer $[Sys]$ as seen in the previous section. Below we list the most important aspects of the model in no particular order.

- Each node has a separate message queue, the message queue is emptied on recovery of a process. (This is done implicitly by the framework and is not visible in the algorithm code.) This means that it is possible for a process to receive messages intended for another process (if another process sends a message to a predecessor of a process, since only the host is used as the 'address'), but it will not inherit messages from its predecessor. The queue model is further discussed in Sect. 5.3.
- Messages are received in a FIFO-manner. I.e., there is no such thing as pattern matching on messages in the message queue, which is often used in Erlang. Note, this is a property of the algorithm, and the full Erlang message handling would require a more complicated model.



Note: elec_1, elec_2, wait and norm also have a self-loop with 'On NormQ'

Figure 3: State space overview

- State vectors are kept orthogonally to the set of processes. Instead of having one state vector for each process, we have a set of state arrays indexed by host id in which the state is stored. (`acks, downs, status, pendacks, elids, ldrs` and `queues`) are the state arrays. For example $Acks[\texttt{host}(Pid)]$, returns the set of acknowledged processes for Pid. The `queues`-array is not used explicitly by the algorithm, instead we use a more general send (**!**) operator. The state arrays and their types are listed in Fig. 4. The state arrays correspond to the variables listed in Tab. 1; except for *Pid*, which is instead a local variable. I.e., each process has a *Pid*-variable, where its process identifier is stored. The *Pid* variable is frequently used to index the state arrays.
- The send operator (**!**) is overloaded. The left-hand side can either be a host identifier or a set of host identifiers. In the second case a message is sent to every member of the set.
- In order to reason about the global state we also have two global sets; *alive* which contains all alive process identifiers, and *pids* which contains all processes that have ever communicated with another process.
- All *state variables* are expressed as either *global sets*, or as an index in a *state*

array. Using some rewriting, both the sets and the arrays and their updates can be expressed in terms of two simple predicates (`setIn` and `index`) and ordinary logic symbols.

- The distinction between process identifiers and node identifiers is necessary. We need to be able to distinguish between different incarnations of a process on a particular node. Each incarnation will have the same node identifier, but a unique process identifier.
- The operators $\oplus=$ and $\ominus=$ are set addition and removal. For example, $Set \oplus= Elem$ is equivalent to $Set := Set \cup \{Elem\}$.
- The **On** `NormQ` and **On** `NotNorm` parts of the model are not needed for the safety property we are trying to prove. (But are essential for the liveness properties of the algorithm.) They are part of the protocol to allow for processes that have missed the election to start another election, where they are included.

5.1 The Initial State

We also need to express what the initial state of the system is. The natural way to express the initial state is to say that no processes are alive at that time and that all sets, message queues, etc. are empty. A first-order logic formulation of this is:

$$
\begin{array}{l}
\forall Pid.(\\
\quad ((Pid \notin \texttt{pids}) \\
\quad \wedge (Pid \notin \texttt{alive}) \\
\quad \wedge (\texttt{queues}(\texttt{host}(Pid)) = \texttt{q\_nil}) \\
\quad \wedge (\texttt{ldrs}[\texttt{host}(Pid)] = \texttt{zero}) \\
\quad \wedge (\texttt{elids}[\texttt{host}(Pid)] = \texttt{nil}) \\
\quad \wedge (\texttt{pendacks}[\texttt{host}(Pid)] = \texttt{zero}) \\
\quad \wedge (\texttt{acks}[\texttt{host}(Pid)] = \texttt{setEmpty}) \\
\quad \wedge (\texttt{downs}[\texttt{host}(Pid)] = \texttt{setEmpty}) \\
\quad \wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_1})) \\
)
\end{array}
$$

5.2 Axiomatization

To represent some aspects of the model in a comprehensible way we have used axiomatizations for natural numbers, message queues, arrays and sets. In the following sections we give some short description of the axiomatizations and explanation for choices we have made.

Host Identifiers

Host identifiers (the result of applying the `host()`-function to process identifiers) could be seen as (a restricted kind of) natural numbers. Host identifiers are

```
local (pid) Pid
state_array (set host) Acks, Downs
state_array (host) Pendacks, Ldrs
state_array (status) Status
state_array (pid) Elids
global (set pid) Pids, Alive

On recovery :
 StartStage1()

On Halt<T : pid>:
 Downs[host(Pid)] ⊖= host(T)
 Elids[host(Pid)] := T
 Status[host(Pid)] := wait
 Pids ⊕= Pid
 host(T) ! Ack<T,Pid>

On Ack<T : pid,J : pid>:
 if (Status[host(Pid)]=elec_2 ∧ Elids[host(Pid)]=T ∧
     host(J)=Pendacks[host(Pid)]) then
   Acks[host(Pid)] ⊕= host(J)
   ContinueStage2()
 fi

On Down<T : pid>:
 if host(T)< host(Pid) then
   Downs[host(Pid)] := Downs[host(Pid)] ∪ {host(T)}
   if (Status[host(Pid)]=norm ∧ Ldrs[host(Pid)]=host(T)) ∨
      (Status[host(Pid)]=wait ∧ host(T)=host(Elids[host(Pid)])) then
     Alive ⊖= Pid
     StartStage1()
   else
     if (Status[host(Pid)]=elec_1 ∧ lesser(host(Pid)) ⊆ Downs[host(Pid)]) then
       StartStage2()
     fi
   fi
 else
   if (Status[host(Pid)]=elec_2 ∧ host(T)=Pendacks[host(Pid)]) then
     Downs[host(Pid)] ⊕= host(T)
     ContinueStage2()
   fi
 fi
```

Figure 4: Crashing process – Stoller part 1 of 3

ordered (all comparisons are made using a less than or equal ($\leq$) predicate) and it is possible to take the successor of a host identifier to yield the next host. The axioms for $\leq$ are listed in Appendix A.4.4.

```
On Ldr<T : pid>:
 if (Status[host(Pid)]=wait ∧ Elids[host(Pid)]=T) then
   Ldrs[host(Pid)] := host(T)
   Status[host(Pid)] := norm
 fi

On NormQ<T : pid>:
 if Status[host(Pid)]≠ norm then
   host(T) ! NotNorm<T>
 fi

On NotNorm<T : pid>:
 if (Status[host(Pid)]=norm ∧ Ldrs[host(Pid)]=host(Pid) ∧
     Elids[host(Pid)]=T) then
   Alive ⊖= Pid
   StartStage1()
 fi

Periodically :
 if (Status[host(Pid)]=norm ∧ Ldrs[host(Pid)]=host(Pid)) then
   greater(host(Pid)) ! NormQ<Elids[host(Pid)]>
 fi

On crash :
 Alive ⊖= Pid
 Pids ⊕= Pid
 Alive ! Down<Pid>
```

Figure 5: Crashing process – Stoller part 2 of 3

Sets and Arrays

Most of the state variables are stored in arrays, and the algorithm uses a number of sets to store state data. As mentioned earlier, by using rewriting, the sets and the arrays and their updates can be expressed in terms of two simple predicates (`setIn` and `index`) plus ordinary logic symbols. Therefore, the set of axioms for sets and arrays are very limited. In fact, the only axiom is one stating that the empty set has no elements.

Message Queues

Message queues are in effect only a list of messages. However, in the algorithm we manipulate the queue at both ends. Therefore we decided to use an aximatization including both `cons` and `snoc`. That is, we can easily access both the first and the last element of the queue. We also have a simple way to describe that a particular element is at the head of the queue: $Queue = \texttt{cons}(Elem, Queue')$, and addition of a new element at the end of a queue: $Queue' = \texttt{snoc}(Queue, Elem)$. The axioms for message queues are found in Appendix A.4.3. There are also axioms that define the `twice` predicate; a predicate that is true when a particular message is

```
Procedure StartStage1 :
 Pid := newPid()
 Alive ⊕= Pid
 Elids[host(Pid)] := Pid
 Status[host(Pid)] := elec_1
 Downs[host(Pid)] := ∅
 if host(Pid)=1 then
  StartStage2()
 fi

Procedure StartStage2 :
 Status[host(Pid)] := elec_2
 Acks[host(Pid)] := ∅
 Pendacks[host(Pid)] := host(Pid)
 ContinueStage2()

Procedure ContinueStage2 :
 if Pendacks[host(Pid)]< nbr_proc then
  Pendacks[host(Pid)] := Pendacks[host(Pid)] + 1
  Pids ⊕= Pid
  Pendacks[host(Pid)] ! Halt<Pid>
 else
  Ldrs[host(Pid)] := host(Pid)
  Status[host(Pid)] := norm
  Acks[host(Pid)] ! Ldr<Pid>
 fi
```

Figure 6: Crashing process – Stoller part 3 of 3

repeated in a message queue. These axioms are presented in Appendix A.4.6.

Problem Specific Axioms

There are also quite a few problem specific axioms that are needed. Most of them express (non-)equality properties, one such example is: $\forall X, Y.(\text{m\_Down}(X) \neq \text{m\_Ldr}(Y))$, which says that a Down-message is never equal to a Ldr-message. These axioms are presented in Appendix A.4.1.

There are also axioms that define the `ordered` predicate. `ordered`(*Queue*) is true exactly when the messages in *Queue* are ordered with respect to the process identifiers that the messages contains. For this algorithm it is necessary to consider the order between `Halt-, Ack-, Ldr-` and `Down`-messages. The axioms are presented in Appendix A.4.3.

5.3 Message Queue Model

In earlier work, we have seen that the model for message passing is vital for the ability to discover errors in the algorithm (or implementation) [ACS05]. The difference between the message queue models lies in how messages are delivered. In the simpler model, used in several different projects [FGN+03, ABS04], messages

are delivered in exactly the order in which they are sent. This simplification is true (in an Erlang system) as long as all involved processes are run in a single run-time system, but not in the general distributed case. The effect of this is that possible re-orderings of messages are missed. In the paper 'A more accurate semantics for distributed Erlang' [SF07] a detailed description of the differences is presented.

In this work we have, in order to keep the proof reasonably simple, chosen not to use the more complex view of message passing. It should be rather straightforward to take the message re-ordering into account in a first-order logic model. (For example, we could use a model with a message queue for each pair of communicating processes.) However, it is unclear how much more complicated the proof would get.

We have made another simplification in order to keep the proof at a reasonable level of atomicity. Each code block (On recovery, On Msg, etc.) is an atomic block. This might seems strange at first, but each block contains only one *visible* action. (Visible actions in this setting means exactly message(s) being sent.) Therefore, to an outside observer (the other processes) either the whole block is executed or not, and there is no need for smaller atomic blocks. In two places the atomic blocks are an oversimplification; namely in the `else`-branch of **On** `Ack` and in **On** `NotNorm`, where messages are sent to each node in a set. A more accurate model should allow for the process to crash in the middle of this operation. (Note, in **On** `Crash` messages are also sent to a set of processes, but here atomicity is not an issue. This is the way we model failure detectors.)

It remains future work to prove the safety property also for models with the details discussed in this section included.

6 The Main Invariant

We want to establish the critical safety property "There is never more than one leader" for the leader election algorithm. To prove this we formulated the following invariant for the system:

$$
\begin{array}{l}
\forall \mathit{Pid}, \mathit{Pid2}.(\\
\quad (((\mathit{Pid} \in \texttt{alive}) \\
\qquad \wedge (\mathit{Pid2} \in \texttt{alive}) \\
\qquad \wedge (\texttt{status}[\texttt{host}(\mathit{Pid})] = \texttt{norm}) \\
\qquad \wedge (\texttt{ldrs}[\texttt{host}(\mathit{Pid})] = \texttt{host}(\mathit{Pid})) \\
\qquad \wedge (\texttt{status}[\texttt{host}(\mathit{Pid2})] = \texttt{norm}) \\
\qquad \wedge (\texttt{ldrs}[\texttt{host}(\mathit{Pid2})] = \texttt{host}(\mathit{Pid2}))) \\
\qquad \rightarrow (\mathit{Pid} = \mathit{Pid2}) \\
\quad) \\
)
\end{array}
$$

The invariant basically reads: whenever two different processes (*Pid* and *Pid2*) both consider themselves as the leader (that is, they are alive, in state `norm` and have themselves as the `ldr`) *Pid* is exactly equal to *Pid2*.

Unfortunately, this invariant is too weak to be inductively provable. If we look at the proof procedure, the invariant obviously holds in the initial state. In

the initial state no processes are alive and thus the antecedent of the invariant is trivially false. However, the step-case does not hold. It is not hard to see that we can construct a system state from which a step takes the system into a state where the invariant does not hold. Nevertheless, this state is non-reachable and it is up to us to strengthen the invariant to prove this. In total 88 additional invariants are needed to prove the main invariant, where some invariants specify more general properties about the system and other invariants prevent only a very specific system situation. All 89 invariants are presented together with a small explanation in the Appendix.

7 Implementation

All implementation work has been made in Haskell [HHJW07]. We have implemented an embedded language in which crashing processes can be expressed. We have also implemented a rewrite-system that applies a predicate transformer $[Sys]$ to a first-order logic formula (representing an invariant). The rewrite-system is used in the system transition step described in Sect. 4. Some of the generated proof obligations are not solvable by the automated theorem provers without manual interaction. We have implemented some primitives to make this manual first-order logic formula interaction easier.

7.1 Embedded Language for Algorithm

We have implemented an embedded language in Haskell that is suitable for describing algorithms including crashing participants. Since the goal of the work was not this language in particular, some support is missing and the instruction set is very targeted for this particular application. As an example there is no parser, meaning that all code is written directly as a large data type expression. However, there is a pretty printer, and the code in Fig. 4, 5 and 6 is mostly machine generated. The language is fairly simple, reflecting the requirements by the proof procedure; the main constructions are if-then-else, assignment, and various set- and array-operations.

7.2 The []-Operator

The implementation of the predicate transformer (the []-operator) is straightforward. For each program construction there is a direct mapping to its consequences for a logic formula as described in Sect. 4.

7.3 Proof Tactics

Some of the generated first-order logic problems can not be solved in a reasonable time (< 3 minutes). This does not necessarily mean that the problems are particularly hard. It could just as well be that their structure is non-optimal for the theorem prover. One example of such a non-optimal structure is the originally not very complex expression from the algorithm:

$\texttt{lesser}(\texttt{host}(Pid)) \subseteq (\texttt{downs}[\texttt{host}(Pid)] \cup \{\text{host}(Pid2)\})$

The expression means that the process (Pid) has (with the addition of host($Pid2$)) received a Down-message from all process in its `lesser`-set (all processes with a higher priority). This is processed by the Set-simplification into a rather complex expression:

$$
\begin{array}{l}
\forall V.(\\
\quad (((\texttt{s}(\texttt{zero}) \leq V) \ \wedge \ (\texttt{host}(Pid) > V)) \\
\qquad \rightarrow ((\, V \in \texttt{downs}[\texttt{host}(Pid)]) \\
\qquad\quad \vee \, (\, V = \texttt{host}(Pid2))) \\
\quad) \\
)
\end{array}
$$

The problem is that the theorem prover does not assign a special meaning to this structure. It simply treats the structure as a composed formula, not as a single unit. Given the problem structure where the invariants are present both in the antecedent and the consequent, it is often crucial to the proof that one instance of this structure could be matched with another. Therefore, one whole class of problems becomes significantly easier (for an automated theorem prover) if the complex expression above is substituted for the term:

$\forall V.\ \texttt{lesserIsSubsetOfDown}(V, Pid, Pid2)$.

Another set of hard to solve logic problems can be helped by giving a hint to the theorem prover. We have done this by specifying a particular case-split to be used.

We have implemented a set of operations on logic formulas, including the substitution indicated above and a safe case-split. The case-split is implemented in such a way that it is first proved that the cases are total before the case-split is used. I.e., given the problem $A \rightarrow B$ and the case-split $C_1 \vee C_2$, we first prove that $A \rightarrow (C_1 \vee C_2)$ before trying to prove both $A \wedge C_1 \rightarrow B$ and $A \wedge C_2 \longrightarrow B$. It is non-trivial to find out which case-splits to try, but with some experience it becomes easier.

8 Proof

In this section we present some data from the proof procedure, and we also show an example of a proof obligation.

8.1 Statistics

The algorithm as presented in Fig. 4, 5 and 6, consists of 9 sections (six different **On** `msg`, **On crash**, **On recovery** and **Periodically**). In order to prove it correct, we have used 89 invariants. For each invariant, each of the nine program sections have been applied, generating a number of logic problems (the number depends on the program and the invariant structure). In total, after performing as much case-splitting as possible ($A \longrightarrow (B_1 \wedge B_2)$ is split into $A \longrightarrow B_1$ and $A \longrightarrow B_2$), there are 13563 first-order logic problems that should be solved.

13510 of the problems are solved by running a selection of automated theorem provers (described below) with carefully planned timeouts. Solving these 13510

problems require no user interaction, and takes around 5 hours. (5h09m on a fairly modest, P4 2.4GHz, workstation with 1GB of RAM.)

8.2 Tactics

To solve the remaining 53 problems we need to apply some proof tactics as described above. Approximately 40 of these problems are trivially solved by applying the rewrite described in section 7.3 where a complicated set-expression is rewritten to a single representative function.

The other problems are solved by specifying more or less complicated case-splits and hints. One specific example of such a case-split defined in Haskell is shown in Fig. 7. The cases are defined for a particular host ($\texttt{host}(X)$) where $\texttt{host}(X)$ is either in a particular set, equal to another host or larger than this other host.

```
tac_inv40_subprob114 =
  do fsf <- mapM parseFOF
              ["setIn(host(X),index(downs,host(V2)))",
               "host(X) = host(V3)",
               "leq(host(V3),host(X))"]
     return (CaseSplit fsf)
```

Figure 7: case-split example

8.3 Theorem Provers

In the verification we have used four different automated theorem provers: E-prover, Vampire, SPASS and Equinox. They have different reasoning frameworks as well as different strategies implemented. Thus they work well on different categories of problems, and most problems are quickly provable by at least one of the theorem provers. A comparison of the theorem provers is presented in Fig. 8 and 9. In Fig. 8 the number of instances solved with a timeout of 600 seconds is presented and in Fig. 9 the total number of solved problems for each theorem prover is presented for two different timeouts (100 s and 600 s).

E-prover

E-prover [Ep] is based on the equational superposition calculus. E-prover, in contrast to many other provers, implements a purely equational paradigm and simulates non-equational inferences via appropriate equality inferences. E-prover has shared term rewriting (where many possible equational simplifications are carried out in a single operation), several efficient term indexing data structures for speeding up inferences and advanced inference literal selection strategies. E-prover is one of the stronger existing theorem provers and its automatic strategy usually performs rather well.

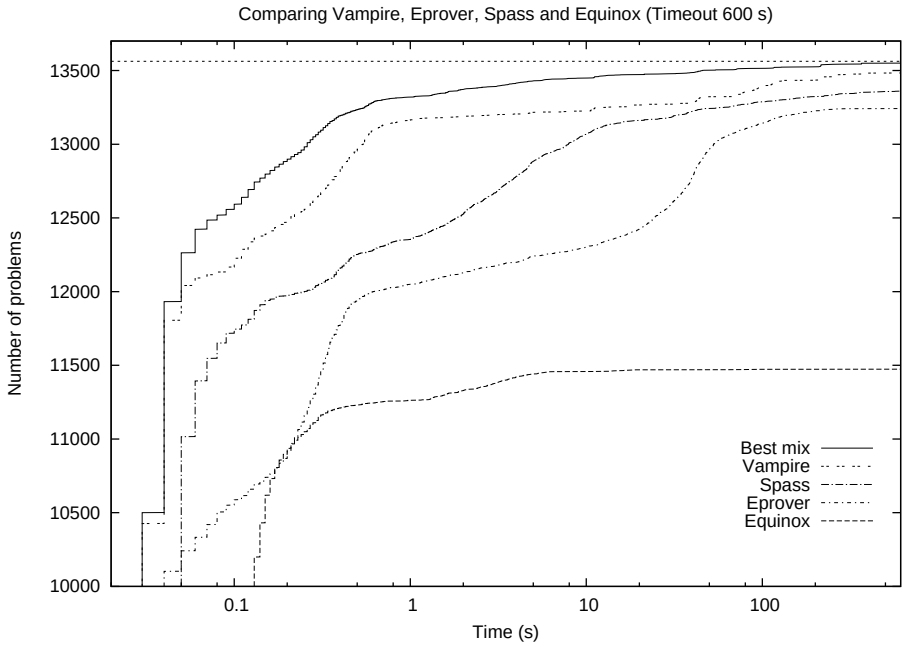


Figure 8: Comparing theorem provers, timeout 600 s.

Vampire

Vampire [Vam] is an automatic theorem prover for first-order classical logic developed by Prof. Andrei Voronkov previously together with Dr. Alexandre Riazanov. Vampire has a fast kernel, implementing a calculus of ordered binary resolution which uses superposition to handle equality. Vampire implements a wide selection of simplification and redundancy removal techniques such as subsumption, tautology deletion, rewriting, etc. Vampire has won the CASC [SS06] competition no less than seven times, and is considered the strongest general purpose automated theorem prover.

SPASS

SPASS [SPA] is a saturation-based automated theorem prover for first-order logic with equality. SPASS features a combination of the superposition calculus with specific inference/reduction rules for sorts (types). While not being as strong as Vampire, there are quite a few invariants where SPASS is performing well.

Equinox

Equinox [Equ] is an experimental new theorem prover for pure first-order logic with equality. The strategy of Equinox is to find ground proofs of the input theory, this is done by solving successive ground instantiations of the theory

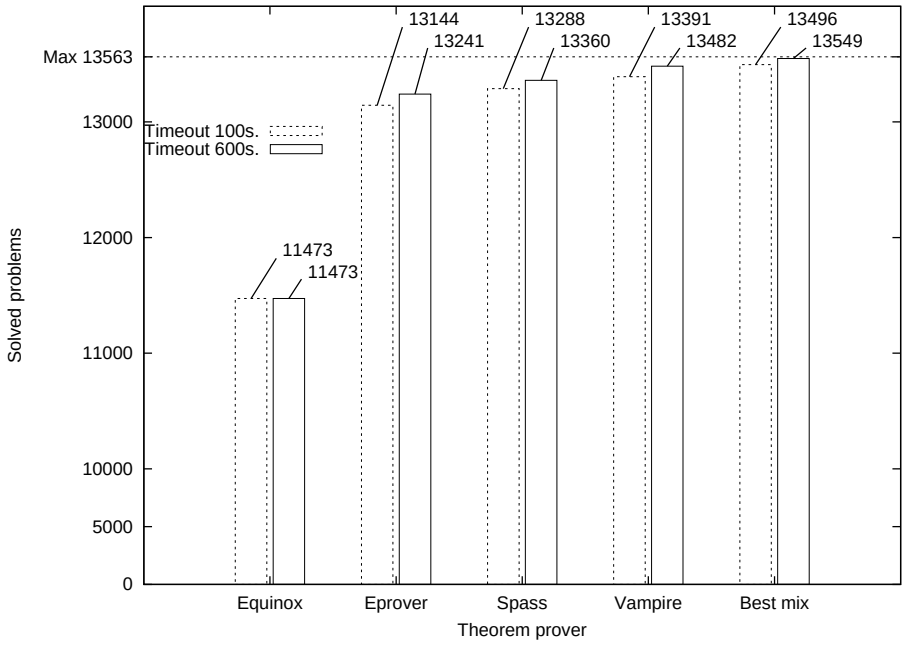


Figure 9: Comparing theorem provers, total number of solved problems

using an incremental SAT-solver. Equinox is not a very strong theorem prover in general, but there are problems solved by Equinox that are very hard for the other theorem provers.

8.4 Proof Example

To further explain the proof procedure we also present a concrete example. Here we have the main invariant (also presented in Sect. 6):

$$
\begin{array}{l}
\forall \mathit{Pid2}, \mathit{Pid}.(\\
\quad (((\mathit{Pid} \in \texttt{alive}) \\
\qquad \wedge (\mathit{Pid2} \in \texttt{alive}) \\
\qquad \wedge (\texttt{status}[\texttt{host}(\mathit{Pid})] = \texttt{norm}) \\
\qquad \wedge (\texttt{ldrs}[\texttt{host}(\mathit{Pid})] = \texttt{host}(\mathit{Pid})) \\
\qquad \wedge (\texttt{status}[\texttt{host}(\mathit{Pid2})] = \texttt{norm}) \\
\qquad \wedge (\texttt{ldrs}[\texttt{host}(\mathit{Pid2})] = \texttt{host}(\mathit{Pid2}))) \\
\qquad \rightarrow (\mathit{Pid} = \mathit{Pid2}) \\
\quad) \\
)
\end{array}
$$

If we apply the system **On** `Ldr`<*T*> (See Fig. 5) to the invariant using the implementation of the []-operator, we end up (after case-splitting, described below) with the following formula:

$$
\begin{array}{l}
\forall X, W, V.(\\
\quad ((\texttt{queues}(\texttt{host}(X)) = \texttt{cons}(\texttt{m\_Ldr}(T), V)) \\
\qquad \rightarrow ((X \in \texttt{alive}) \\
\qquad\quad \rightarrow (((\texttt{elids}[\texttt{host}(X)] = T) \\
\qquad\qquad \wedge (\texttt{status}[\texttt{host}(X)] = \texttt{wait})) \\
\qquad\qquad \rightarrow \forall \mathit{Pid},\ \mathit{Pid2}.(\\
\qquad\qquad\quad ((\texttt{host}(X) = \texttt{host}(\mathit{Pid})) \\
\qquad\qquad\qquad \wedge (\texttt{host}(X) \neq \texttt{host}(\mathit{Pid2}))) \\
\qquad\qquad\qquad \rightarrow (((\mathit{Pid} \in \texttt{alive}) \\
\qquad\qquad\qquad\qquad \wedge \mathit{Pid2} \in \texttt{alive} \\
\qquad\qquad\qquad\qquad \wedge \texttt{host}(T) = \texttt{host}(\mathit{Pid}) \\
\qquad\qquad\qquad\qquad \wedge \texttt{ldrs}[\texttt{host}(\mathit{Pid2})] = \texttt{host}(\mathit{Pid2}) \\
\qquad\qquad\qquad\qquad \wedge \texttt{status}[\texttt{host}(\mathit{Pid2})] = \texttt{norm}) \\
\qquad\qquad\qquad\qquad \rightarrow (\mathit{Pid} = \mathit{Pid2}) \\
\qquad\qquad\qquad\quad) \\
\qquad\qquad\qquad) \\
\qquad\qquad\quad) \\
\qquad\qquad) \\
\qquad\quad) \\
\qquad) \\
\quad) \\
)
\end{array}
$$

This is not the complete resulting problem, since some case splitting has taken place, namely on the if-statement which is true here and also on $\texttt{host}(X) = \texttt{host}(\mathit{Pid})$ and $\texttt{host}(X) \neq \texttt{host}(\mathit{Pid2})$. We see that some substitutions have been made, for example $\texttt{ldrs}[\texttt{host}(\mathit{Pid})] = \texttt{host}(\mathit{Pid})$ has been replaced by $\texttt{host}(T) = \texttt{host}(\mathit{Pid})$. We can also see that the sub-expression $\texttt{status}[\texttt{host}(\mathit{Pid})] = \texttt{norm}$ is completely missing. This is because $\texttt{status}[\texttt{host}(\mathit{Pid})]$ has been substituted for $\texttt{norm}$ and $\texttt{norm} = \texttt{norm}$ has been simplified away.

8.5 Proof Procedure Summary

Most of what is described in this report is automatic. To summarize the manual work, the following five components is the necessary input:

- The initial state of the system
- The invariant(s)
- The description of the system transitions
- The invariant subsets (named P_i in Sect. 4)
- Manual case-splitting or other manual interaction might be needed for some (hard) proof obligations

It is a repetitive process to come up with the necessary invariants as well as the invariant subsets, while the initial state and the transitions of the system

is not as volatile. Given these five components, proof obligations are automatically generated. The generated proof obligations are automatically case-split, and some re-writing (mainly for sets and arrays) is performed. The resulting proof obligations are given to the automated theorem provers described in Sect. 8.3.

For some of the proof obligations the theorem provers are not powerful enough, and further manual interaction is necessary as described in Sect. 7.3.

9 Conclusions

In this document we have presented a formalization and a proof of correctness for the Bully Algorithm (as presented by Stoller [Sto97]). By doing this we have confirmed that the informal reasoning by Garcia-Molina [GM82], actually carries over (at least as far as the conclusion that the algorithm is correct) to (our model of) Stoller's formulation of the algorithm. Our model accurately describes Stoller's version of the Bully Algorithm. However, it remains future work to improve the model further; for example, to also include the more accurate message queue model and improve the modeling of messages sent to a set of recievers.

We have also described a proof procedure that hopefully can be more generally applicable than for just this proof. The proposed way to model communicating processes and the proposed axiomatizations could certainly be useful in correctness proofs for other distributed and fault-tolerant algorithms.

References

[Abr96] J-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ABS04] T. Arts, C. Benac Earle, and J.J. Sánchez Penas. Translating Erlang to mCRL. In *Fourth International Conference on Application of Concurrency to System Design*, p. 135–144, Hamilton (Ontario), Canada, June 2004. IEEE Computer Society.

[ACS05] Thomas Arts, Koen Claessen, and Hans Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. In *Lecture Notes in Computer Science*, vol. Vol. 3395, p. 140 – 154, Feb 2005.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[Ep] E-prover. http://www.eprover.org.

[Equ] Equinox. http://www.cs.chalmers.se/~koen/folkung/.

[FGN+03] Lars-Åke Fredlund, Dilian Gurov, Thomas Noll, Mads Dam, Thomas Arts, and Gennady Chugunov. A verification tool for Erlang. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):405–420, August 2003.

[FS07] L-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed function al programming language. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2007.

[GM82] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, January 1982.

[HHJW07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, San Diego, California, June 2007.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hol03] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003. ISBN: 0-321-22862-6.

[SF07] Hans Svensson and Lars-Åke Fredlund. A more accurate semantics for distributed Erlang. In *Erlang '07: Proceedings of the 2007 SIGPLAN workshop on Erlang Workshop*, p. 43–54, New York, NY, USA, 2007. ACM.

[SPA] SPASS. http://spass.mpi-sb.mpg.de/index.html.

[SS06] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[Sto97] S.D. Stoller. Leader election in distributed systems with crash failures. Technical Report 481, Computer Science Dept., Indiana University, May 1997. Revised July 1997.

[Vam] Vampire. http://www.vampire.fm.

[Wor96] J.B. Wordsworth. *Software Engineering with B*. Addison-Wesley, 1996.

Paper 6

Finding Counter Examples in Induction Proofs

This paper was written together with Koen Claessen. It was published at 'The Second International Conference on Tests and Proofs' (TAP), in Prato, Italy, April 2008. The paper included here has a few minor corrections and is also typeset in a slightly different style.

Finding Counter Examples in Induction Proofs

Koen Claessen[1], and Hans Svensson[1]

[1] Chalmers University of Technology, Göteborg, Sweden
{koen,hanssv}@cs.chalmers.se

Abstract

This paper addresses a problem arising in automated proof of invariants of transition systems, for example transition systems modelling distributed programs. Most of the time, the actual properties we want to prove are too weak to hold inductively, and auxiliary invariants need to be introduced. The problem is how to find these extra invariants. We propose a method where we find *minimal counter examples* to candidate invariants by means of *automated random testing* techniques. These counter examples can be inspected by a human user, and used to adapt the set of invariants at hand. We are able to find two different kinds of counter examples, either indicating (1) that the used invariants are too strong (a concrete trace of the system violates at least one of the invariants), or (2) that the used invariants are too weak (a concrete transition of the system does not maintain all invariants). We have developed and evaluated our method in the context of formally verifying an industrial-strength implementation of a fault-tolerant distributed leader election protocol.

1 Introduction

This paper gives a partial report on our experiences on using (semi-)automated theorem proving to formally verify safety properties of an industrial-strength implementation of a fault-tolerant leader election protocol in the programming language Erlang [19].

Leader election is a basic technique in distributed systems; a fixed set of processes has to determine a special process, the *leader*, among them. There is one basic safety property of such algorithms ("there should never be more than one leader"), and one basic liveness property ("eventually there should be one leader"). In *fault-tolerant* leader election, processes can die and be restarted at any point in time (during or after the election), making the problem immensely tricky.

Erlang is a language for distributed programming originally developed for implementing telecommunication systems at Ericson [3, 2]. A key feature of the systems for which Erlang was primarily designed is fault-tolerance; Erlang has therefore built-in support for handling failing processes.

The implementation of the leader election algorithm we verified was developed by us, after we had uncovered some subtle bugs in an earlier existing implementation using testing techniques [4]. Our new implementation is based on an adaptation of a standard fault-tolerant leader election algorithm by Stoller [18] and is now a standard library in Erlang. In our implementation, we had to make some changes to Stoller's original algorithm because of the way processes communicate in Erlang (via asynchronous message passing over unbounded channels) and the

way fault-tolerance is handled in Erlang (a process can monitor another process, in which case it receives a special message when the other process dies).

From our previous experience, we knew that it is extremely hard to get these kinds of algorithms right. Indeed, we started by extensively testing the new implementation using our testing techniques [4], leading to our increased confidence in the correctness of the implementation. However, we had some reasons to be cautious. Firstly, our implementation was based on an adaptation of Stoller's original algorithm, so even if Stoller's algorithm were correct, our adaptation of it might not be. Secondly, Stoller never gives a formal proof of correctness in his paper [18]. His algorithm is in turn an adaptation of a classical leader election algorithm (called "The Bully Algorithm") by Garcia-Molina, which in turn only has been proven correct in the paper in a very informal way [12]. Stoller claims that his modifications are so minor that giving a new proof is not needed: *"The proofs that the Bully$_{FD}$ Algorithm satisfies SLE1 and SLE2 are very similar to the proofs of Theorems A1 and A2 in [GM82] and are therefore omitted."*

When we decided to formally verify our implementation, we first tried a number of different model checking methods (among others SPIN [13] and our own model checker McErlang [11]). Unfortunately, these could only be used for extremely small and unconvincing bounds on the number of processes, sizes of message queues, and number of times processes can die. This is partially due to the huge state space generated by the combination of asynchronous message passing and fault-tolerance.

The alternative we eventually settled on was to prove invariants of the system inductively by means of automated first-order logic theorem proving. Here, we model the implementation as an abstract transition system, and express the properties we want to prove as invariants on the states of the transition system. The reasons we chose this approach were (1) using first-order logic allowed us to prove the implementation correct for any number of processes, using unbounded message queues and an unbounded number of occurring faults, and (2) automated first-order theorem provers are relatively autonomous, in principle only requiring us to interact with the verification process at the level of choosing the invariants.

The main obstacle in this approach is that, most often, the (relatively small) set of invariants one is interested in establishing is not inductively provable. This means that the original set of invariants has to be strengthened by changing some of the invariants or by augmenting the set with new invariants, until the set is strong enough to be inductive. Very often, this is a non-trivial and labour-intensive task. In our case, we started with one invariant ("there should not be more than one leader") and we ended up with a set of 89 invariants. This is the sense in which we call our method *semi-automated*; if the right set of invariants is picked (manually), the proof is carried out automatically. Thus, the user of the method does not have to carry out proofs, but only has to formulate proof obligations.

The task of finding the right set of invariants is not only non-trivial, but can also be highly frustrating. The reason is that it is very easy for a user, in an attempt to make the set of invariants stronger, to add properties to the set which are in fact not invariants. When certain invariants can not be proven, the first-

order theorem provers we use do not in general provide any reason as to why this is the case, leaving the user in the dark about what needs to be done in order to get the proof through.

We identified 4 different reasons for why a failed proof of a given invariant occurs: (1) the invariant is invalid, i.e. there exists a path from the initial state to a state where the invariant is falsified, (2) the invariant is valid, but too weak, i.e. it indeed holds in all reachable states, but it is not maintained by the transition relation, (3) the invariant is valid and is maintained by the transition relation, but the current axiomatization of the background theories is too weak, and (4) the invariant is valid and should be provable, but the theorem prover at hand does not have enough resources to do so.

The remedies for being in each of these cases are very different: For (1), one would have to weaken the invariant at hand; for (2) one would have to strengthen it; for (3) one would have to come up with extra axioms or induction principles; for (4) one would have to wait longer or break the problem up into smaller bits.

Having a concrete counter example to a proof attempt would show the difference between cases (1), (2) and (3). Thus, having a way of finding counter examples would greatly increase the productivity of the proposed verification method. Providing counter models to first-order formulas (or to formulas in more complex logics) is however an undecidable problem.

We have developed two novel methods, based on random property-based testing using the automated testing tool QuickCheck [9], that, by automatically reusing the invariants as test generators and test oracles, can automatically and effectively find counter examples of categories (1) and (2). Finding counter examples of category (3) remains future work.

Establishing inductive invariants is a very common method for verifying software (in particular in object-oriented programs, see for example [5, 21]). We believe that the methods for finding counter examples in this paper can be adapted to other situations than verifying distributed algorithms.

The contributions of this paper are:

- A classification of different categories of counter examples in the process of establishing inductive invariants using a theorem prover
- Two methods for finding two of the most common categories of counter examples based on random testing
- An evaluation of the methods in the context of the verification of an industrial-strength implementation of a leader election protocol

The rest of the paper is organized as follows. The next section explains the method of verification we use in more detail. Section 3 explains the testing techniques we use. Section 4 reports on the results of our method in the verification of the leader election implementation. Section 5 concludes.

2 Verification Method

In this section, we describe the basic verification method we use to prove invariants. The method is quite standard; an earlier description of the method in the context of automated first-order logic reasoning tools can be found in [8]. The system under verification and the invariants are described using three components:

- A predicate $Init$ describing the initial state,
- A predicate Inv describing the invariant,
- A predicate transformer $[Sys]$ that abstractly describes one transition of the system.

For the predicate transformers, we borrow notation also used in dynamic logic [5] and the B-method [1, 21]. For a program S and a post-condition Q, we write $[S]Q$ to be the weakest pre-condition for S that establishes Q as a post-condition. This in turn means that we can write

$$P \rightarrow [S]\, Q$$

which has the same meaning as the Hoare triple $\{P\}S\{Q\}$; in all states where P holds, making the transition described by S leads to states where Q holds.

The language we use to describe Sys is very simple. The three most important constructs are assignments, conditionals, and non-deterministic choice. The definition of predicate transformers we use is completely standard, and we will only briefly discuss the concepts here. For more details, the reader can consult [21]. Here are the definitions for the predicate transformers for assignments, conditionals, and non-deterministic choice, respectively.

$$\begin{array}{rcl} [x := e]\, P & = & P\{e/x\} \\ [\textbf{if}\, Q\, \textbf{then}\, S\, \textbf{else}\, T]\, P & = & (Q \rightarrow [S]P) \wedge (\neg Q \rightarrow [T]P) \\ [S \mid T]\, P & = & [S]P \wedge [T]P \end{array}$$

Establishing Inv as an invariant amounts to proving the following two statements:

$$\begin{array}{rcl} Init & \rightarrow & Inv \\ Inv & \rightarrow & [Sys]\, Inv \end{array}$$

In practice, Inv is really a conjunction of a number of smaller invariants:

$$\begin{array}{rcl} Init & \rightarrow & Inv_1 \wedge Inv_2 \wedge \cdots \wedge Inv_n \\ Inv_1 \wedge Inv_2 \wedge \cdots \wedge Inv_n & \rightarrow & [Sys]\, (Inv_1 \wedge Inv_2 \wedge \cdots \wedge Inv_n) \end{array}$$

The above two proof obligations are split up into several sub-obligations; for the initial states, we prove, for all i, several obligations of the form:

$$Init \rightarrow Inv_i$$

$\forall Pid, Pid2.($
$\quad (\mathbf{elem}(\texttt{m\_Halt}(Pid),$
$\qquad\qquad \texttt{queue}(\texttt{host}(Pid2)))$
$\qquad \rightarrow (\texttt{host}(Pid2) > \texttt{host}(Pid))$
$\quad)$
$)$

The invariant states that `Halt`-messages are only sent to processes with lower priority: If there is a `Halt`-message from *Pid* in the queue of $\texttt{host}(Pid2)$, then *host(Pid2)* is larger than *host(Pid)*. (Hosts with low numbers have high priority.)

Figure 1: Example invariant

For the transitions, we prove, for all i, several obligations of the form:

$$\left(\bigwedge_{j \in P_i} Inv_j \right) \rightarrow [Sys]\, Inv_i$$

So, for each invariant conjunct Inv_i, we have a subset of the invariants P_i that we use as a pre-condition for establishing Inv_i. Logically, we can use all invariants Inv_j as pre-condition, but in practice the resulting proof obligations would become too large to be manageable by the theorem provers we use. Also, from a proof engineering point of view, it is good to "localize" dependencies, so that when the set of invariants changes, we only have to redo the proofs for the obligations that were involved in the invariants we changed. (Note that the set P_i can actually include the invariant Inv_i itself.)

To simplify the problems as much possible, we also use an aggressive case splitting strategy, in the same way as described in [8]. Thus each of the above proof obligations is proved in many small steps.

In Fig. 1 we show an example of an invariant. The function $\texttt{host}(p)$ returns the host for a given process p, the predicate $\mathbf{elem}(m, q)$ is true if a message m is present in a message queue q. In this example we have an incoming message queue $\texttt{queue}(h)$ for each host h. (This simplification from having a message queue per process is possible since there is only one process alive per host.)

2.1 Failed Proof Attempts

This paper deals with the problem of what to do when a proof attempt of one of the proof obligations fails. Let us look at what can be the reason for a failed proof attempt when proving the proof obligations related to a particular candidate invariant Inv_i. We can identify 4 different reasons:

(1) The candidate invariant Inv_i is not an invariant of the system; there exists a reachable state of the system that falsifies Inv_i.

(2) The candidate invariant Inv_i actually is an invariant of the system, but it is not an inductive invariant. This means that there exists an (unreachable) state where all invariants in the pre-condition set P_i of Inv_i are true, but after a transition, Inv_i is not true. This means that the proof obligation for the transition for Inv_i cannot be proven.

(3) The candidate invariant Inv_i actually is an invariant of the system, and it is an inductive invariant. However, our background theory is not strong enough to establish this fact. The background theory contains axioms about message queues, in what order messages arrive, what happens when processes die, etc. If these are not strong enough, the proof obligation for the transition for Inv_i cannot be proven.

(4) The proof obligations are provable, but the theorem prover we use does not have enough resources, and thus a correctness proof cannot be established.

When a proof attempt for a proof obligation fails, it is vital to be able to distinguish between these 4 cases. The remedies in each of these cases are different:

For (1), we have to *weaken* the invariant Inv_i, or perhaps remove it from the set of invariants altogether.

For (2), we have to *strengthen* the set of pre-conditions P_i. We can do this by strengthening some invariants in P_i (including Inv_i itself), or by adding a new invariant to the set of invariants and to P_i.

For (3), we have to *strengthen* the background theory by adding more axioms.

For (4), we have to *simplify* the problem by for example using explicit case-splitting, or perhaps to give the theorem prover more time.

2.2 Identifying the Categories

How can we identify which of the cases (1)-(4) we are in? A first-order logic theorem prover does not give any feedback in general when it does not find a proof. Some theorem provers, including the ones we used (Vampire [20], E-prover [16], SPASS [10], and Equinox [7]) do provide feedback in certain cases, for example in the form of a finite-domain counter model or a saturation, but this hardly ever happens in practice.

One observation that we can make is that for cases (1)-(3), there exist counter examples of different kinds to the proof obligations.

For (1), the counter example is a concrete trace from the initial state to the reachable state that falsifies the invariant Inv_i.

For (2), the counter example is a concrete state that makes the pre-conditions P_i true, but after one transition the invariant Inv_i does not hold anymore.

For (3), the counter example is a concrete counter model that makes the background theory true but falsifies the proof obligation. This counter model must be a *non-standard* model of the background theory, since the proof obligation is true for every standard model (which is implied by the fact that no concrete counter example of kind (2) exists).

We would like to argue that, if the user were given feedback consisting of (a) the category of counter example above, and (b) the concrete counter example, it would greatly improve productivity in invariant-based verification.

In the next section, we show how we can use techniques from random testing to find counter examples of type (1) and (2) above. We have not solved the problem of how to find counter examples of type (3), which remains future work. (This is an unsolvable problem in general because of the semi-decidability of first-order logic.) Luckily, cases (1) and (2) are most common in practice, because, in our

experience, the background theory stabilizes quite quickly after the start of such a project.

We would like to point out a general note on the kind of counter examples we are looking for. Counter examples of type (1) are counter examples in a logic in which we can define transitive closure of the transition relation. This is necessarily a logic that goes beyond first-order logic. This logic for us exists only on the meta-level, since we are merely performing the induction base case and step case with theorem provers that can not reason about induction. Counter examples of type (2) are only counter examples of the induction step (and do not necessarily imply the existence of counter examples of the first kind). In some sense, these can be seen as non-standard counter examples of the logic used in type (1) counter examples. Counter examples of type (3) are also counter examples of the induction step, but they do not follow the intended behavior of our function and predicate symbols, and are therefore non-standard counter examples of the induction step.

3 Finding Counter Examples by Random Testing

This section describes the random testing techniques that we used to find concrete counter examples to the proof obligations.

3.1 QuickCheck

QuickCheck [9] is a tool for performing specification-based random testing, originally developed for the programming language Haskell. QuickCheck defines a simple executable specification logic, in which universal quantification over a set is implemented as performing random tests using a particular distribution. The distribution is specified by means of providing a test data generator. QuickCheck comes equipped with random generators for basic types (Integers, Booleans, Pairs, Lists, etc) and combinator functions, from which it is fairly easy to build generators for more complex data structures.

When QuickCheck finds a failing test case (a test case that falsifies a property), it tries to *shrink* this test case by successively checking if smaller variants of the original failing test case are still failing cases. When the shrinking process terminates, a (locally) minimal failing test case is presented to the user. The user can provide custom shrinking functions that specify what simplifications should be tried on the failing case. This is a method akin to *delta debugging* [22].

For example, if we find a randomly generated concrete trace which makes an invariant fail, the shrinking function says that we should try removing one step from the trace to see if it is still a counter example. When the shrinking process fails, the trace we produce is minimal in the sense that every step in the trace is needed to make the invariant fail. One should note that it is very valuable to have short counter examples; it drastically reduces the time spent on analyzing and fixing the errors found.

3.2 Trace counter examples

A *trace counter example* is a counter example of type (1) in the previous section. We decided to search for trace counter examples in the following manner (this is inspired by 'State Machine Specifications' in [14]). Given a set of participating processes, we can construct an exhaustive list of possible operations (examples of operations could be: process X receives a `Halt`-message, process Y crashes, process Z is started, etc). We constructed a QuickCheck generator that returns a random sequence of operations. To test the invariant we then create the initial state for the system (where all participants are dead and all message queues are empty) and apply the operation sequence. The result is a sequence of states, and in each state we check that the invariant holds.

If a counter example to the invariant is found, shrinking is performed by simply removing some operations. To further shrink a test case we also try to remove one of the participating processes (together with its operations). We illustrate how all of this works with the (trivially incorrect) invariant $\forall Pid.\neg isLeader(Pid)$ (i.e. there is never a leader elected). Formulated in QuickCheck, the property looks as follows:

```
prop_NeverALeader =
  \path -> checkPath leStoller (forAll pid (nott (isLeader pid))) path
```

We use the function `checkPath`, which takes three arguments: a model of an Erlang program (in this case `leStoller`), a first-order formula (the property) and a trace (called `path`), and checks that the given formula is true for all states encountered on the specified path. The QuickCheck property states that the result should be true for all paths. Running QuickCheck yields:

```
*QCTraceCE> quickCheck prop_NeverALeader
*** Failed! Falsifiable (after 3 tests and 3 shrinks):
Path 1 [AcStart 1]
```

The counter example is a path involving one process (indicated by "`Path 1`", and one step where we start that process (indicated by "`AcStart 1`"), and clearly falsifies the property. (The leader election algorithm is such that if there is only a single participant, it is elected immediately when it is started.) This counter example has been shrunk, in 3 shrinking steps, from an initial, much larger, counter example. The steps it went through, removing unnecessary events, in this case were:

```
Path 1 [AcOnMsg 1 AcLdr,AcOnMsg 1 AcDown,AcOnMsg 1 AcAck,AcOnMsg 1 AcHalt,
        AcStart 1,AcStart 1,AcOnMsg 1 AcNormQ,AcPer 1]
Path 1 [AcStart 1,AcStart 1,AcOnMsg 1 AcNormQ,AcPer 1]
Path 1 [AcStart 1,AcStart 1]
Path 1 [AcStart 1]
```

Here, "`AcOnMsg` p m" indicates that process p receives a message of type m. The different message types ("`AcLdr`", "`AcDown`", "`AcAck`", etc.) are part of the internal details of Stoller's leader election protocol [18] and are not explained here.

Being able to quickly generate locally minimal counter examples to candidate invariants greatly improved our productivity in constructing a correct set of invariants.

3.3 Induction step counter examples

Step counter examples are counter examples of type (2). To find step counter examples is more challenging. Step counter examples can be expected when the stated invariant holds, but its pre-conditions are too *weak* to be proved. The proof fails in the step case, that is there exists a (**non**-reachable) state s such that the invariant is true in s, but false in some state s', such that $s' \in next(s)$. The difference from trace counter examples is that we are now looking for non-reachable states, which are significantly harder to generate in a good way.

Our first, very naive, try was to simply generate completely random states, and check if the proof obligation can be falsified by these. We implemented this strategy by constructing a random generator for states and tried to use QuickCheck in the straightforward way. However, not surprisingly, this fails miserably. The reason is that it is very unlikely for a randomly generated state to fulfill all pre-conditions of the proof obligation for the transition. Other naive approaches, such as enumerating states in some way, do not work either, since the number of different states are unfeasibly large, even with very small bounds on the number of processes and number of messages in message queues.

The usual way to solve this in QuickCheck testing is to make a *custom generator* whose results are very likely to fulfill a certain condition. However, this is completely unpractical to do by hand for an evolving set of about 90 invariants.

Instead, we implemented a *test data generator generator*. Given a first-order formula ϕ, our generator-generator automatically constructs a random test data generator which generates states that are very likely to fulfill ϕ. So, instead of manually writing a generator for each invariant Inv_i, we use the generator-generator to generate one. We then use the resulting generator in QuickCheck to check that the property holds.

Our generator-generator, given a formula ϕ, works as follows. Below, we define a process, called `adapt` that, given a formula ϕ and a state s, modifies s so that it is more likely to make ϕ true. The generator first generates a completely random state s, and then successively *adapts* s to ϕ a number of times. The exact number of times can be given as a parameter.

The `adapt` process works as follows. Given a formula ϕ and a state s, we do the following:

1 Check if s fulfills ϕ. If so, then we return s.

2 Otherwise, look at the structure of ϕ.

- If ϕ is a conjunction $\phi_1 \wedge \phi_2$, recursively adapt s to the left-hand conjunct ϕ_1, and then adapt the result to the right-hand conjunct ϕ_2.
- If ϕ is a disjunction $\phi_1 \vee \phi_2$, randomly pick a disjunct ϕ_i, and adapt s to it.

- If ϕ starts with a universal quantifier $\forall x \in S.\psi(x)$, S will be concretely specified by the state s. We construct a big explicit conjunction $\bigwedge_{x \in S} \psi(x)$, and adapt s to it.
- If ϕ starts with an existential quantifier $\exists x \in S.\psi(x)$, construct a big explicit disjunction $\bigvee_{x \in S} \psi(x)$, and adapt s to it.
- If ϕ is a negated formula, push the negations inwards and adapt s to the non-negated formula.
- If ϕ is a (possibly negated) atomic formula, change s so that the atomic formula is true, if we know how to (see below). Otherwise, just return s.

Quantifiers in ϕ always quantify over things occurring in the state s, for example the set of all processes, or the set of all processes currently alive, etc. When adapting s to ϕ, these sets are known, so we can create explicit conjunctions or disjunctions instead of quantifiers.

When randomly picking a disjunct, we let the distribution be dependent on the size of the disjuncts; it is more likely here to pick a large disjunct than a small disjunct. This was added to make the process more fair when dealing with a disjunction of many things (represented as a number of binary disjunctions).

Finally, we have to add cases that adapt a given state s to the atomic formulae. The more cases we add here, the better our adapt function becomes. Here are some examples of atomic formulae occurring in ϕ, and how we adapt s to them:

- "message queue $q1$ is empty", in this case we change the state s such that $q1$ becomes empty;
- "process $p1$ is not alive", in this case we remove $p1$ from the set of alive processes in s;
- "queue $q1$ starts with the message $Halt$", in this case we simply add the message $Halt$ to the queue $q1$.

Note that there is no guarantee that an adapted state satisfies the formula. For example, when adapting to a conjunction, the adaption process of the right-hand conjunct might very well undo the adaption of the left-hand conjunct. It turns out that successively adapting a state to a formula several times increase the likelihood of fulfilling the formula. There is a general trade-off between adapting a few states many times or adapting many states fewer times. The results of our experiments suggest that adapting the same state 4-8 times is preferable (Sect. 4).

The final property we give to QuickCheck looks as follows; remember the problem

$$\left(\bigwedge_{j \in P_i} Inv_j \right) \rightarrow [Sys]\, Inv_i$$

and let `invs` be the left hand side of the implication, `inv` is Inv_i and `applySys` corresponds to the []-operation:

```
prop_StepProofObligation invs inv sys =
  \state ->
    forAll (adapt formula state) $ \state' ->
      checkProperty formula state'
  where formula = and (nott inv' : invs)
        inv'    = applySys sys inv
```

This can be read as: For all states s, and for all adaptions s' of that state s to the proof obligation, the proof obligation should hold. The function `adapt` is our implementation of the adapt generator-generator, and `checkProperty` checks if a given formula is true in a given state. Remember that we want to find a counter example state, that is why we try to adapt the state so that the pre-conditions (`invs`) are fulfilled but `inv'` is not.

The experimental results are discussed in the next section.

4 Results

In this section we present some results from the usage of search for counter examples in the verification of the leader election algorithm. Since the data comes from only one verification project it might not be statistically convincing, but it should be enough to give some idea of how well the search for counter examples works in practice.

4.1 Trace Counter Examples

To illustrate the effectiveness of trace counter examples we first show one particular example. In Fig. 2 we see an invariant A that was added to the set of pre-conditions in order to be able to prove another invariant B (i.e. this was the action taken after a failed proof attempt in category 2, as described in Sect. 2.1). The original invariant B was easily proved after this addition, however we could not prove the new invariant A. After several days of failed proof attempts, we managed to (manually) find a counter example. The counter example was really intricate, involving four different nodes and a non-trivial sequence of events.

With this unsatisfying experience in fresh memory, we were eager to try the trace counter example finder on this particular example. The result was very positive, the counter example was quickly found (in the presented run after 170 tests), and we could quickly verify that it was equivalent to the counter example that we found manually. The result of the QuickCheck run on this example is presented in Fig. 3.

The counter example consists of a `Path` value. From this value we can conclude that the counter example involves four processes. We can also see the sequence of operations leading to a state where the invariant is falsified. This sequence contains five process starts (`AcStart`), three process crashes (`AcCrash`) and two receives of `Down`-messages by process number 3 (`AcOnMsg`). It is interesting to see that the fourth process is never started, and never actually does anything, nevertheless it must be present in order to falsify the invariant (or else the shrinking would have removed it).

$$
\begin{array}{l}
\forall Pid, Pid2, Pid3.((\\
\quad((Pid \in \texttt{alive})\\
\qquad \wedge\ \mathbf{elem}(\texttt{m\_Down}(Pid2),\\
\qquad\qquad\qquad \texttt{queue}(\texttt{host}(Pid)))\\
\qquad \wedge\ (\texttt{lesser}(\texttt{host}(Pid)) \subseteq\\
\qquad\qquad (\texttt{down}[\texttt{host}(Pid)] \cup \{\texttt{host}(Pid2)\}))\\
\qquad \wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_1}))\\
\qquad \rightarrow \neg((\texttt{pendack}[\texttt{host}(Pid3)] > \texttt{host}(Pid))\\
\qquad\qquad \wedge\ (Pid3 \in \texttt{alive})\\
\qquad\qquad \wedge\ (\texttt{status}[\texttt{host}(Pid3)] = \texttt{elec\_2}))\\
\quad)\\
)
\end{array}
$$

Whenever a process (*Pid*) is alive, in the first election phase (`elec_1`) and it receives a `Down`-message such that *Pid* has received `Down`-messages from everyone with higher priority (that is the hosts in the set `lesser(host(`*Pid*`))`). Then no other process (here *Pid3*) is alive, in the second election phase and having communicated with *Pid* (i.e. having a `pendack` value larger than `host(`*Pid*`)`).

Figure 2: A broken invariant

```
*** Failed! Falsifiable (after 170 tests and 30 shrinks):
Path 4 [AcStart 2,AcStart 3,AcCrash 2,AcStart 1,AcCrash 1,
        AcOnMsg 3 AcDown,AcStart 2,AcOnMsg 3 AcDown,AcStart 1,AcCrash 1]
```

Figure 3: Trace counter example

Evaluation of Trace Counter Examples

Although the verification process was complicated, we did not have very many badly specified invariants around to test with. The presented example was the most complicated and in total we had some five or six *real* 'broken' invariants to test with. (All of them produced a counter example.) To further evaluate the trace counter example search in a more structural way, we used a simplistic kind of *mutation testing*. We took each invariant and negated (or if it was already negated, removed the negation) all sub-expressions occurring on the left hand side of an implication. Thereafter we tried the trace counter example search for each of the mutated invariants.

In total we generated 272 mutated invariants. We tried to find a trace counter example for each, and succeeded in 187 cases (where we randomly generated 300 test cases for each invariant). However, we should not expect to find a trace counter example in all cases, since some of the mutated invariants are still true invariants. Manual inspection of 10 of the 85 ($272-187=85$) failed cases revealed only two cases where we should expect to find a counter example. (A re-run of the two examples with a limit of 1000 generated tests was run, and a counter example was found in both cases.)

4.2 Induction Step Counter Examples

To illustrate how the inductive step counter examples could be used we use the invariant presented below as an example. This invariant was actually the last invariant that was added in order to complete the proof of the leader election algorithm. The invariant specifies a characteristic of the acknowledgement messages sent during election.

$$
\begin{array}{l}
\forall \mathit{Pid}, \mathit{Pid2}, \mathit{Pid3}.(\\
\quad (((\mathit{Pid2} \neq \mathit{Pid3}) \\
\qquad \wedge \mathbf{elem}(\texttt{m\_Ack}(\mathit{Pid}, \mathit{Pid2}), \\
\qquad\qquad \texttt{queue}(\texttt{host}(\mathit{Pid}))) \\
\qquad \wedge (\texttt{host}(\mathit{Pid2}) = \texttt{host}(\mathit{Pid3}))) \\
\qquad \rightarrow \neg \mathbf{elem}(\texttt{m\_Ack}(\mathit{Pid}, \mathit{Pid3}), \\
\qquad\qquad \texttt{queue}(\texttt{host}(\mathit{Pid}))) \\
\quad) \\
)
\end{array}
$$

If *Pid2* and *Pid3* are two different processes at the same host, and an `Ack`-message from *Pid2* to *Pid* is in *Pid*'s queue, then there can not also be an `Ack`-message in the queue of *Pid* sent by *Pid3* to *Pid*.

Figure 4: Invariant for step counter example example

The first proof attempt included invariants 3, 14 and 15 (which are also invariants that specify properties about `Ack`-messages), i.e. we tried to prove $(Inv_3 \wedge Inv_{14} \wedge Inv_{15} \wedge Inv_{89}) \rightarrow [Sys]\, Inv_{89}$. This proof attempt fails, and if we search for an induction step counter example we get the following state:

```
State with 2 processes:
* Alive: {(2,3),(2,5)}
* Pids:  {(2,3)}
[ Process: (1,2)
  Status: norm   Elid: (2,3)  Ldr: 1  Pendack: 2
  Queue: [Ack (1,2) (2,3)]
  Acks: {}   Down: {},

  Process: (2,3)
  Status: wait   Elid: (1,2)  Ldr: 2  Pendack: 2
  Queue: [Halt (1,2)]
  Acks: {}   Down: {}]
```

The system state consists of two sets `alive` (that contains the process identifiers of all processes currently alive) and `pids` (that contains all process identifiers ever used). A process identifier is implemented as a pair of integers. Furthermore, the individual state of each process is also part of the system state. Each process state has a number of algorithm-specific variables (`Status`, `Elid`, etc.), and an incoming message queue.

In the counter example we see that the second process has a `Halt`-message from the first process in its queue at the same time as there is an `Ack`-message in the queue of the first process. That means that in the next step the second process could acknowledge the `Halt`-message, and thus create a state in which the invariant is falsified. Indeed such a situation can not occur, and we actually already had an invariant (with number 84) which stated exactly this. Therefore, if we instead try to prove: $(Inv_3 \wedge Inv_{14} \wedge Inv_{15} \wedge Inv_{84} \wedge Inv_{89}) \longrightarrow [Sys]\, Inv_{89}$ we are successful.

Evaluation of Step Counter Examples

In the verification of the leader election algorithm we used 89 sub-invariants which were proved according to the scheme

$$(Inv_1 \wedge Inv_2 \wedge \cdots \wedge Inv_k) \longrightarrow [Sys]\, Inv_1$$

Since the automated theorem provers are rather sensitive to the problem size, we put some effort into creating *minimal* left hand sides of the implication. That is, we removed the sub-invariants that were not needed to prove a particular sub-invariant.

Therefore, a simple way to generate evaluation tests for the step counter example search is to remove yet another sub invariant from the left hand side and thus get a problem which in most cases (the minimization was not totally accurate) is too weak to be proved in the step case. Thus, we generate a set of problems like

$$(Inv_1 \wedge Inv_2 \wedge \cdots Inv_{k-1} \wedge Inv_{k+1} \wedge \cdots \wedge Inv_n) \longrightarrow [Sys]\, Inv_1$$

and evaluate the step counter example search on this set of problems.

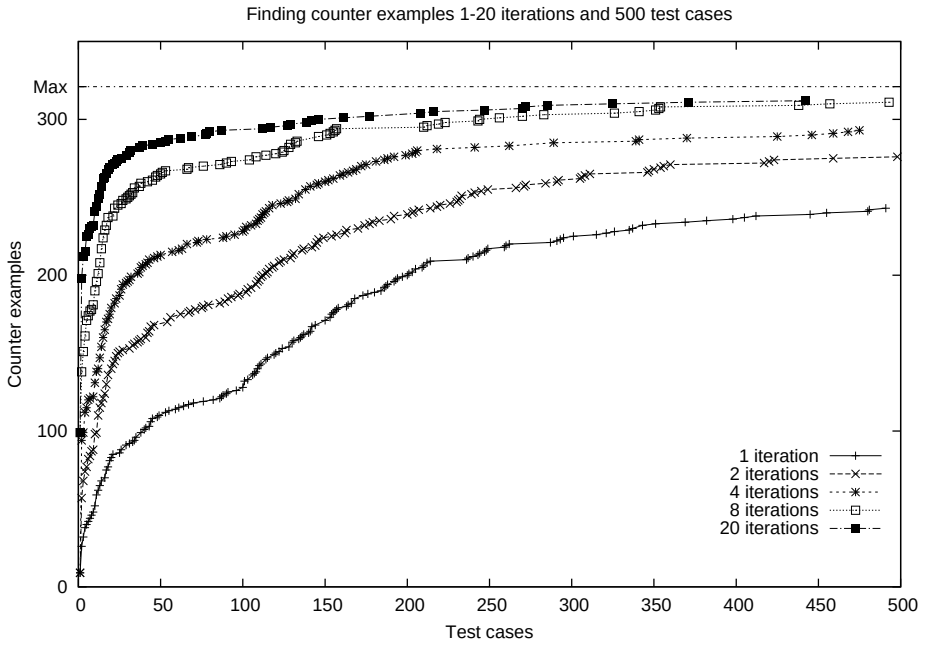


Figure 5: Step counter example results

In this way, the 89 proof obligations were turned into 351 problems to test the step counter example search with. More careful analysis revealed that 30 of the problems were actually still provable, thus leaving 321 test cases. The result of running the step counter example search in QuickCheck with 500 test cases for each problem, and a varying number of adapt rounds, is presented in Fig. 5.

In the figure we see that with only one iteration of adapt we find a counter example for around 75% of the tested problems. By increasing the number adapt rounds, we find a counter example for 97% of the tested problems within 500 test cases.

In reality, case-splitting [8] turned these 321 into 1362 smaller problems of which 524 are provable. The results of running the step counter example search in QuickCheck for each of these smaller problems are presented in Fig. 6. The results are quite similar to the results in the earlier figure.

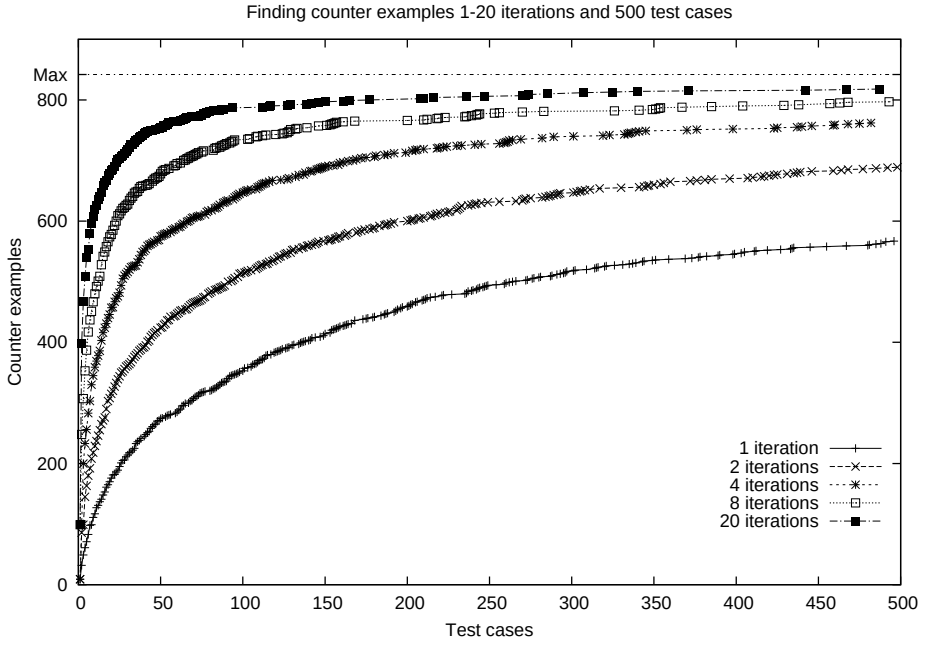


Figure 6: Step counter example results

Our conclusion is that this way of finding counter examples is remarkably effective, especially keeping in mind that the counter example search we presented is a fully automatic and a very cheap method. Running QuickCheck for a failed proof attempt takes only from a few seconds, sometimes up to a few minutes.

Another important aspect is the quality of the counter examples; i.e. given an induction step counter example, how hard is it to figure out how to strengthen the invariant to make it provable. Of course this is hard to measure, and any judgement here is highly subjective. We randomly selected some of the found counter examples and inspected them more carefully. In most cases it was easy to find out which sub-invariant to add, which was the original purpose of the method.

Interestingly, in some examples, the counter example indicated that a certain sub-invariant was missing, which was different from the sub-invariant we had removed. (Remember, we generated the tests by removing one sub-invariant

from already proved examples.) It turned out that we could actually prove the problem by either using the removed sub-invariant *or* the sub-invariant suggested by the counter example. For example: from the (already proved) problem $(Inv_4 \wedge Inv_7 \wedge Inv_8) \longrightarrow [Sys]Inv_8$ we removed Inv_4. This resulted in a counter example, which indicated that adding Inv_2 would probably make it possible to prove the sub-invariant. Indeed the problem $(Inv_2 \wedge Inv_7 \wedge Inv_8) \longrightarrow [Sys]\, Inv_8$ could be proved. The reason for this is that Inv_2 and Inv_4 were partially overlapping. The conclusion must nevertheless be that an induction step counter example is most often very useful.

5 Discussion and Conclusion

We have identified different categories of reasons why proof attempts that establish inductive invariants may fail, and developed a method that can identify 2 of these categories by giving feedback in terms of a concrete counter example.

We would like to argue that the results show that this is a useful method; very often counter examples are found when they should be found, and they are easy to understand because of the (local) minimality. The method is also very cheap, once the system is set up, it does not take much time or resources to run 300 random tests. Every time we make changes to the set of invariants, a quick check can be done to make sure no obvious mistakes have been made.

For related work, just like pure first-order logic theorem provers, interactive theorem proving systems usually do not provide feedback in terms of a counter example either. ACL2 [15] provides feedback by producing a log of the failed proof attempt. While sometimes useful, we would like to argue that feedback in terms of counter examples (and in terms of different kinds of counter examples) is more directly useful for a user. In some work in the context of rippling [17], a failed proof attempt is structurally used to come up with an invariant for while-loops in imperative programs.

The interactive higher-order logic reasoning system Isabelle comes with a version of QuickCheck [6]. However, there is no control over generators or shrinking present in this version. The work presented here can possibly be integrated with Isabelle by extending their QuickCheck with the necessary features.

Some might argue that the main problems presented in the paper disappear when moving to a reasoning system that supports induction, for example ACL2 or a higher-order theorem prover. However, in such systems it is still useful to have a notion of different reasons why inductive proofs fail, and the three types of counter examples (1), (2) and (3) are just as useful in such systems.

For future work, we are looking to further reduce the gap between problems where proofs are found and problems where counter examples are found. We are currently working to augment a theorem prover to also give us feedback that can be used to identify categories (3) and (4). For category (3), an approximation of a non-standard counter model is produced, for category (4), the theorem prover can tell why it has not found a proof yet.

Moreover, we want to study liveness more closely, and integrate liveness checking (and finding counter examples) in the overall verification method.

Finally, to increase the applicability of our work, we would like to separate out the different parts of our current system; the counter example finding from the Erlang-specific things, and the leader-election-specific axioms and invariants from the general Erlang axioms.

References

[1] J-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J. Armstrong. *Programming Erlang – Software for a Concurrent World*. The Pragmatic Programmers, http://books.pragprog.com/titles/jaerlang, 2007.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

[4] Thomas Arts, Koen Claessen, and Hans Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. In *Lecture Notes in Computer Science*, volume Vol. 3395, p. 140 – 154, Feb 2005.

[5] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer-Verlag, 2007.

[6] T. Berghofer, S.; Nipkow. Random testing in isabelle/hol. *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, p. 230–239, 28-30 Sept. 2004.

[7] Koen Claessen. Equinox, a new theorem prover for full first-order logic with equality. Presentation at Dagstuhl Seminar 05431 on Deduction and Applications, October 2005.

[8] Koen Claessen, Reiner Hähnle, and Johan Mårtensson. Verification of hardware systems with first-order logic. In *Proc. of Problems and Problem Sets Workshop (PaPS)*, 2002.

[9] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, p. 268–279, New York, NY, USA, 2000. ACM.

[10] C. Weidenbach et al. SPASS: An automated theorem prover for first-order logic with equality. http://spass.mpi-sb.mpg.de.

[11] L-Å. Fredlund and H. Svensson. McErlang: A model checker for a distributed functional programming language. In *Proc. of International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, 2007.

[12] Hector Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, January 1982.

[13] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, September 2003. ISBN: 0-321-22862-6.

[14] John Hughes. QuickCheck testing for fun and profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *LNCS*, p. 1–32. Springer-Verlag, Berlin Heidelberg, 2007.

[15] Matt Kaufmann and J Strother Moore. ACL2 - A Computational Logic / Applicative Common Lisp. http://www.cs.utexas.edu/users/moore/acl2/.

[16] Stephan Schulz. The e equational theorem prover. http://eprover.org.

[17] Jamie Stark and Andrew Ireland. Invariant discovery via failed proof attempts. In *Proceedings, 8th International Workshop on Logic Based Program Synthesis and Transformation*, 1998.

[18] S.D. Stoller. Leader election in distributed systems with crash failures. Technical Report 481, Computer Science Dept., Indiana University, May 1997. Revised July 1997.

[19] Hans Svensson and Thomas Arts. A new leader election implementation. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, p. 35–39, New York, NY, USA, 2005. ACM Press.

[20] Andrei Voronkov. Vampire. http://www.vampire.fm.

[21] J.B. Wordsworth. *Software Engineering with B*. Addison-Wesley, 1996.

[22] Andreas Zeller. Isolating cause-effect chains from computer programs. In *SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, p. 1–10, New York, NY, USA, 2002. ACM.

Appendix

A.1 Appendix Overview

Most of the material in the appendix is generated directly from the input to the proof procedure, which is annotated with explaining text. Sect. A.2 and A.3 describes the predicate symbols and function symbols used in the first-order logic formulas. Sect. A.4 lists all the used axioms, divided into roughly the same categories as Sect. 5.2. Lastly, in Sect. A.5, all the 89 invariants are listed.

A.2 Predicates

- **elem/2** – elem(Msg,Queue) is true if the message is present in Queue.
- **isSubset/2** – Set1 $\subseteq$ Set2, is true if Set1 is a subset of Set2.
- **leq/2** – N1 $\leq$ N2, is true if N1 is less than or equal to N2.
- **ordered/1** – ordered(Queue), is true if the messages in the Queue is ordered according to Pids.
- **setIn/2** – Elem $\in$ Set, is true if Elem is a member of Set.
- **twice/2** – twice(Msg, Queue), is true if Msg is present twice in Queue.

A.3 Functions/Constant Symbols

A.3.1 State-arrays - indexed by host

- **ldr/0** – Ldr-array, stores the host of the current leader.
- **acks/0** – Acks-array, stores the acks set for each host.
- **down/0** – Down-array, stores the down set for each host.
- **elid/0** – Elid-array, stores the election id for each host.
- **status/0** – Status-array, stores the status for each host.
- **pendack/0** – Pendack-array, stores the host that the process is waiting for an answer from.
- **queue/1** – Queue-array, stores the receive queues for each host.

A.3.2 Natural numbers

- **zero/0** – The constant number zero.
- **s/1** – The successor function
- **nbr_proc/0** – The number of participating processes

A.3.3 State-names

- **elec_1/0** – Election state 1, waiting for higher prioritized process.
- **elec_2/0** – Election state 2, waiting for process with lower priority.
- **wait/0** – Waiting state, waiting for Ldr-message from new leader.
- **norm/0** – Normal state, either the process is the leader or knows the leader.

A.3.4 Global sets

- **alive/0** – A set that stores all processes that are currently alive.
- **pids/0** – A set that stores all processes that has ever existed.

A.3.5 Messages

- **m_Halt/1** – Halt-message, contains the sender.
- **m_Ack/2** – Ack-message, contains the sender and the acknowledged process.
- **m_Ldr/1** – Ldr-message, contains the leader pid.
- **m_Down/1** – Failure message, contains the failing pid.
- **m_NormQ/1** – Question for normality, is replied to if a process is not in state norm.
- **m_NotNorm/1** – A response to a NormQ-message, contains the process that asked NormQ.

A.3.6 Set functions

- **setUnion/2** – The normal set union operation, written $\cup$.
- **setAdd/2** – Adds an element to a set, written $\oplus$.
- **setDel/2** – Deletes an element from a set, written $\ominus$.
- **lesser/1** – Given a host, returns a set of all hosts with higher priority

A.3.7 Array functions

- **index/2** – Indexes an array, written array[*ind*].

A.3.8 Host function

- **host/1** – Given a pid the function returns the host of the pid

A.4 Axioms

A.4.1 Axioms – stoller

- $\forall P, Q.($
  $((\mathtt{s}(\mathtt{host}(P)) = \mathtt{host}(Q))$
  $\rightarrow (\mathtt{host}(P) \neq \mathtt{host}(Q))$
  $)$
  $)$
- $\forall P.((\mathtt{s}(\mathtt{zero}) \leq \mathtt{host}(P)))$
- $(\mathtt{s}(\mathtt{zero}) \leq \mathtt{nbr\_proc})$
- $\forall P.((\mathtt{host}(P) \leq \mathtt{nbr\_proc}))$
- $(\mathtt{elec\_1} \neq \mathtt{elec\_2})$
- $(\mathtt{elec\_1} \neq \mathtt{wait})$
- $(\mathtt{elec\_1} \neq \mathtt{norm})$
- $(\mathtt{elec\_2} \neq \mathtt{wait})$
- $(\mathtt{elec\_2} \neq \mathtt{norm})$
- $(\mathtt{norm} \neq \mathtt{wait})$
- $\forall X, Y, Z.((\mathtt{m\_Ack}(X, Y) \neq \mathtt{m\_Halt}(Z)))$
- $\forall X, Y, Z.((\mathtt{m\_Ack}(X, Y) \neq \mathtt{m\_Down}(Z)))$
- $\forall X, Y, Z.((\mathtt{m\_Ack}(X, Y) \neq \mathtt{m\_NotNorm}(Z)))$
- $\forall X, Y, Z.((\mathtt{m\_Ack}(X, Y) \neq \mathtt{m\_Ldr}(Z)))$
- $\forall X, Y, Z.((\mathtt{m\_Ack}(X, Y) \neq \mathtt{m\_NormQ}(Z)))$
- $\forall X, Y.((\mathtt{m\_NotNorm}(X) \neq \mathtt{m\_Halt}(Y)))$
- $\forall X, Y.((\mathtt{m\_Down}(X) \neq \mathtt{m\_Halt}(Y)))$
- $\forall X, Y.((\mathtt{m\_Down}(X) \neq \mathtt{m\_Ldr}(Y)))$
- $\forall X, Y.((\mathtt{m\_Down}(X) \neq \mathtt{m\_NotNorm}(Y)))$
- $\forall X, Y.((\mathtt{m\_Down}(X) \neq \mathtt{m\_NormQ}(Y)))$
- $\forall X, Y.((\mathtt{m\_NormQ}(X) \neq \mathtt{m\_Halt}(Y)))$
- $\forall X, Y.((\mathtt{m\_Ldr}(X) \neq \mathtt{m\_Halt}(Y)))$
- $\forall X, Y.((\mathtt{m\_Ldr}(X) \neq \mathtt{m\_NormQ}(Y)))$
- $\forall X, Y.((\mathtt{m\_Ldr}(X) \neq \mathtt{m\_NotNorm}(Y)))$
- $\forall X, Y.((\mathtt{m\_NormQ}(X) \neq \mathtt{m\_NotNorm}(Y)))$
- $\forall X, Y.($
  $((X \neq Y) \leftrightarrow (\mathtt{m\_Halt}(X) \neq \mathtt{m\_Halt}(Y)))$
  $)$
- $\forall X, Y.($
  $((X \neq Y) \leftrightarrow (\mathtt{m\_NormQ}(X) \neq \mathtt{m\_NormQ}(Y)))$
  $)$
- $\forall X, Y.($
  $((X \neq Y) \leftrightarrow (\mathtt{m\_NotNorm}(X) \neq \mathtt{m\_NotNorm}(Y)))$
  $)$

- $\forall X, Y.(((X \neq Y) \leftrightarrow (\texttt{m\_Ldr}(X) \neq \texttt{m\_Ldr}(Y))))$
- $\forall X, Y.($
  $((X \neq Y) \leftrightarrow (\texttt{m\_Down}(X) \neq \texttt{m\_Down}(Y)))$
  $)$
- $\forall X1, X2, Y1, Y2.($
  $((X1 \neq X2)$
  $\rightarrow (\texttt{m\_Ack}(X1, Y1) \neq \texttt{m\_Ack}(X2, Y2))$
  $)$
  $)$
- $\forall X1, X2, Y1, Y2.($
  $((Y1 \neq Y2)$
  $\rightarrow (\texttt{m\_Ack}(X1, Y1) \neq \texttt{m\_Ack}(X2, Y2))$
  $)$
  $)$

A.4.2 Axioms – con-sys

- $\forall Pid2, Pid.($
  $((\texttt{host}(Pid) \neq \texttt{host}(Pid2)) \rightarrow (Pid \neq Pid2))$
  $)$
- $(\texttt{nil} \notin \texttt{alive})$

A.4.3 Axioms – cons-snoc

- $\forall X, Q.((\texttt{head}(\texttt{cons}(X, Q)) = X))$
- $\forall X, Q.((\texttt{tail}(\texttt{cons}(X, Q)) = Q))$
- $\forall Y, Q.((\texttt{last}(\texttt{snoc}(Q, Y)) = Y))$
- $\forall Y, Q.((\texttt{init}(\texttt{snoc}(Q, Y)) = Q))$
- $\forall Q.($
  $((Q = \texttt{q\_nil}) \ \vee \ (Q = \texttt{cons}(\texttt{head}(Q), \texttt{tail}(Q))))$
  $)$
- $\forall Q.($
  $((Q = \texttt{q\_nil}) \ \vee \ (Q = \texttt{snoc}(\texttt{init}(Q), \texttt{last}(Q))))$
  $)$
- $\forall X, Q.((\texttt{q\_nil} \neq \texttt{cons}(X, Q)))$
- $\forall Y, Q.((\texttt{q\_nil} \neq \texttt{snoc}(Q, Y)))$
- $\forall X.((\texttt{cons}(X, \texttt{q\_nil}) = \texttt{snoc}(\texttt{q\_nil}, X)))$
- $\forall X, Y, Q.((\texttt{snoc}(\texttt{cons}(X, Q), Y) = \texttt{cons}(X, \texttt{snoc}(Q, Y))))$
- $\forall X.(\neg \mathbf{elem}(X, \texttt{q\_nil}))$
- $\forall X, Y, Q.($
  $(\mathbf{elem}(X, \texttt{cons}(Y, Q)) \leftrightarrow (\mathbf{elem}(X, Q) \ \vee \ (X = Y)))$
  $)$
- $\forall X, Y, Q.($
  $(\mathbf{elem}(X, \texttt{snoc}(Q, Y)) \leftrightarrow (\mathbf{elem}(X, Q) \ \vee \ (X = Y)))$
  $)$

- $\forall X, Y, Q.($
  $(\mathbf{elem}(\texttt{m\_Down}(X), Q)$
  $\leftrightarrow \mathbf{elem}(\texttt{m\_Down}(X), \texttt{snoc}(Q, \texttt{m\_Halt}(Y)))$
  $)$
  $)$
- $\forall X, Y, Q.($
  $(\mathbf{elem}(\texttt{m\_Down}(X), Q)$
  $\leftrightarrow \mathbf{elem}(\texttt{m\_Down}(X), \texttt{snoc}(Q, \texttt{m\_Ldr}(Y)))$
  $)$
  $)$
- $\forall X, Y, Q.($
  $(\mathbf{elem}(\texttt{m\_Down}(X), Q)$
  $\leftrightarrow \mathbf{elem}(\texttt{m\_Down}(X), \texttt{snoc}(Q, \texttt{m\_NotNorm}(Y)))$
  $)$
  $)$
- $\forall X, Y, Q.($
  $(\mathbf{elem}(\texttt{m\_Down}(X), Q)$
  $\leftrightarrow \mathbf{elem}(\texttt{m\_Down}(X), \texttt{snoc}(Q, \texttt{m\_NormQ}(Y)))$
  $)$
  $)$
- $\forall X, Y, Q.($
  $(\mathbf{elem}(\texttt{m\_Ldr}(X), Q)$
  $\leftrightarrow \mathbf{elem}(\texttt{m\_Ldr}(X), \texttt{snoc}(Q, \texttt{m\_NotNorm}(Y)))$
  $)$
  $)$
- $\forall X, Y, Z, Q.($
  $(\mathbf{elem}(\texttt{m\_Ack}(X, Z), Q)$
  $\leftrightarrow \mathbf{elem}(\texttt{m\_Ack}(X, Z), \texttt{snoc}(Q, \texttt{m\_Halt}(Y)))$
  $)$
  $)$
- $\forall X.($
  $(\mathbf{pidElem}(X)$
  $\leftrightarrow \exists Y.($
  $(\exists Z.((X = \texttt{m\_Ack}(Y, Z)))$
  $\vee (X = \texttt{m\_Ldr}(Y))$
  $\vee (X = \texttt{m\_Down}(Y))$
  $\vee (X = \texttt{m\_Halt}(Y)))$
  $)$
  $)$
  $)$
- $\forall X.((\texttt{pidMsg}(\texttt{m\_Halt}(X)) = X))$
- $\forall X.((\texttt{pidMsg}(\texttt{m\_Down}(X)) = X))$
- $\forall X.((\texttt{pidMsg}(\texttt{m\_Ldr}(X)) = X))$
- $\forall X, Y.((\texttt{pidMsg}(\texttt{m\_Ack}(X, Y)) = Y))$
- $\mathbf{ordered}(\texttt{q\_nil})$

- $\forall X.($
  $($**ordered**$(\texttt{cons}(X, \texttt{q\_nil}))$
  $\wedge$ **ordered**$(\texttt{snoc}(\texttt{q\_nil}, X)))$
  $)$
- $\forall X, Q.($
  $($**ordered**$(\texttt{cons}(X, Q))$
  $\leftrightarrow (\forall Y.($
  $((\textbf{elem}(Y, Q)$
  $\wedge$ **pidElem**(X)
  $\wedge$ **pidElem**(Y)
  $\wedge (\texttt{host}(\texttt{pidMsg}(Y)) = \texttt{host}(\texttt{pidMsg}(X))))$
  $\rightarrow (\texttt{pidMsg}(X) \leq \texttt{pidMsg}(Y))$
  $)$
  $)$
  $\wedge$ **ordered**$(Q))$
  $)$
  $)$
- $\forall X, Q.($
  $($**ordered**$(\texttt{snoc}(Q, X))$
  $\leftrightarrow (\forall Y.($
  $((\textbf{elem}(Y, Q)$
  $\wedge$ **pidElem**(X)
  $\wedge$ **pidElem**(Y)
  $\wedge (\texttt{host}(\texttt{pidMsg}(Y)) = \texttt{host}(\texttt{pidMsg}(X))))$
  $\rightarrow (\texttt{pidMsg}(Y) \leq \texttt{pidMsg}(X))$
  $)$
  $)$
  $\wedge$ **ordered**$(Q))$
  $)$
  $)$
- $\forall Q, X, Y.($
  $((\textbf{elem}(\texttt{m\_Down}(Y), Q)$
  $\wedge$ **ordered**$(\texttt{cons}(\texttt{m\_Halt}(X), Q))$
  $\wedge (\texttt{host}(X) = \texttt{host}(Y)))$
  $\rightarrow (X \leq Y)$
  $)$
  $)$
- $\forall Q, X, Y, Z.($
  $((\textbf{elem}(\texttt{m\_Down}(Y), Q)$
  $\wedge$ **ordered**$(\texttt{cons}(\texttt{m\_Ack}(Z, X), Q))$
  $\wedge (\texttt{host}(X) = \texttt{host}(Y)))$
  $\rightarrow (X \leq Y)$
  $)$
  $)$

A.4.4 Axioms – arith

- $\forall X.((\texttt{s}(X) > X))$
- $\forall X.((X \leq X))$

- $\forall X, Y.(((X \leq Y) \ \vee \ (Y \leq X)))$
- $\forall X, Y.((((X \leq Y) \ \wedge \ (Y \leq X)) \leftrightarrow (X = Y)))$
- $\forall X, Y, Z.((((X \leq Y) \ \wedge \ (Y \leq Z)) \rightarrow (X \leq Z)))$
- $\forall X, Y.(((X \leq Y) \leftrightarrow (\mathtt{s}(X) \leq \mathtt{s}(Y))))$
- $\forall X, Y.(((X \leq \mathtt{s}(Y)) \leftrightarrow ((X \leq Y) \ \vee \ (X = \mathtt{s}(Y)))))$

A.4.5 Axioms – sets

- $\forall X.((X \notin \mathtt{setEmpty}))$

A.4.6 Axioms – twice-msg

- $\forall M.(\neg \mathbf{twice}(\mathtt{q\_nil}, M))$
- $\forall M, X.(\neg \mathbf{twice}(\mathtt{cons}(X, \mathtt{q\_nil}), M))$
- $\forall M, Y.(\neg \mathbf{twice}(\mathtt{snoc}(\mathtt{q\_nil}, Y), M))$
- $\forall M, X, Q.($
  $(\mathbf{twice}(\mathtt{cons}(X, Q), M)$
  $\leftrightarrow ((\mathbf{elem}(M, Q) \ \wedge \ (M = X)) \ \vee \ \mathbf{twice}(Q, M))$
  $)$
  $)$
- $\forall M, Y, Q.($
  $(\mathbf{twice}(\mathtt{snoc}(Q, Y), M)$
  $\leftrightarrow ((\mathbf{elem}(M, Q) \ \wedge \ (M = Y)) \ \vee \ \mathbf{twice}(Q, M))$
  $)$
  $)$

A.5 Invariants

stoller_inv1

Main invariant, if two processes (Pid1 and Pid2) are both the leader, then they are equal. I.e. there is never more than one leader.

$\forall Pid, Pid2.($
$(((Pid \in \mathtt{alive})$
$\wedge (Pid2 \in \mathtt{alive})$
$\wedge (\mathtt{status}[\mathtt{host}(Pid)] = \mathtt{norm})$
$\wedge (\mathtt{ldr}[\mathtt{host}(Pid)] = \mathtt{host}(Pid))$
$\wedge (\mathtt{status}[\mathtt{host}(Pid2)] = \mathtt{norm})$
$\wedge (\mathtt{ldr}[\mathtt{host}(Pid2)] = \mathtt{host}(Pid2)))$
$\rightarrow (Pid = Pid2)$
$)$
$)$

Proof needs: [*stoller_inv5*, *stoller_inv6*, *stoller_inv30*, *stoller_inv40*, *stoller_inv50*]

stoller_inv2

System invariant, there is never more than one process alive at a particular host.
$\forall Pid, Pid2.($
$(((Pid \neq Pid2) \ \wedge \ (\texttt{host}(Pid) = \texttt{host}(Pid2)))$
$\rightarrow ((Pid \notin \texttt{alive})$
$\vee \ (Pid2 \notin \texttt{alive}))$
$)$
$)$
Proof needs: []

stoller_inv3

Basic invariant, if a process is in state elec_2 then the value of its pendack variable is greater than the host id.
$\forall Pid.($
$((\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2})$
$\rightarrow (\texttt{pendack}[\texttt{host}(Pid)] > \texttt{host}(Pid))$
$)$
$)$
Proof needs: []

stoller_inv4

Basic invariant, if a process (Pid2) is present in the acks-set of another process (Pid), then host($Pid2$) > host(Pid).
$\forall Pid, Pid2.($
$((\texttt{host}(Pid2) \in \texttt{acks}[\texttt{host}(Pid)])$
$\rightarrow (\texttt{host}(Pid2) > \texttt{host}(Pid))$
$)$
$)$
Proof needs: [*stoller_inv3*]

stoller_inv5

Basic invariant, If a process (Pid2) receives a Ldr-message from another process (Pid), then host($Pid2$) > host(Pid).
$\forall Pid, Pid2.($
$(\textbf{elem}(\texttt{m\_Ldr}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\rightarrow (\texttt{host}(Pid2) > \texttt{host}(Pid))$
$)$
$)$
Proof needs: [*stoller_inv3*, *stoller_inv4*]

stoller_inv6

Basic invariant, if a process (Pid) is alive and in state elec_1 or elec_2 then its election-id (elid) is exactly Pid.

$\forall Pid.($
$(((\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_1})$
$\vee (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\wedge (Pid \in \texttt{alive}))$
$\rightarrow (\texttt{elid}[\texttt{host}(Pid)] = Pid)$
$)$
$)$
Proof needs: [*stoller_inv2*]

stoller_inv7

Basic invariant, If a process (Pid2) receives a Halt-message from another process (Pid), then host(*Pid2*) > host(*Pid*).
$\forall Pid, Pid2.($
$(\textbf{elem}(\texttt{m\_Halt}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\rightarrow (\texttt{host}(Pid2) > \texttt{host}(Pid))$
$)$
$)$
Proof needs: [*stoller_inv3*]

stoller_inv8

Basic invariant, In an Ack-message, the acknowledging pid (here Pid3) always has a higher host-id. I.e. host(*Pid3*) > host(*Pid*).
$\forall Pid, Pid2, Pid3.($
$(\textbf{elem}(\texttt{m\_Ack}(Pid, Pid3), \texttt{queue}(\texttt{host}(Pid2)))$
$\rightarrow (\texttt{host}(Pid3) > \texttt{host}(Pid))$
$)$
$)$
Proof needs: [*stoller_inv7*]

stoller_inv9

Basic invariant, If a Halt-message contains a pid, that pid is also in the set *pids*.
$\forall Pid, Pid2.($
$(\textbf{elem}(\texttt{m\_Halt}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\rightarrow (Pid \in \texttt{pids})$
$)$
$)$
Proof needs: []

stoller_inv10

Basic invariant, If a process (Pid) is alive and in state elec_1, then no Halt-message containing Pid exists.
$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_1}))$
$\rightarrow \neg\textbf{elem}(\texttt{m\_Halt}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$

Proof needs: [*stoller_inv9*, *stoller_inv2*]

stoller_inv11

If a process (Pid) is alive and have sent a Halt-message to another process (Pid2) then its pendack value is $\leq$ host(*Pid2*).

$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge \textbf{elem}(\texttt{m\_Halt}(Pid), \texttt{queue}(\texttt{host}(Pid2))))$
$\rightarrow (\texttt{host}(Pid2) \leq \texttt{pendack}[\texttt{host}(Pid)])$
$)$
$)$

Proof needs: [*stoller_inv10*, *stoller_inv9*, *stoller_inv2*]

stoller_inv12

Basic invariant, If a process (Pid) is alive and in state elec_1, then no Ack-message containing Pid exists.

$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_1}))$
$\rightarrow \neg\textbf{elem}(\texttt{m\_Ack}(Pid, Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv10*, *stoller_inv14*]

stoller_inv13

If a process (Pid) is alive and has received an Ack-message from a process (Pid2), then pendack(*Pid*) $\geq$ host(*Pid2*).

$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge \textbf{elem}(\texttt{m\_Ack}(Pid, Pid2), \texttt{queue}(\texttt{host}(Pid))))$
$\rightarrow (\texttt{host}(Pid2) \leq \texttt{pendack}[\texttt{host}(Pid)])$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv11*, *stoller_inv12*, *stoller_inv14*]

stoller_inv14

Basic invariant, If an Ack-message acknowledges Pid, then Pid is also in the set *pids*.

$\forall Pid, Pid2.($
$(\textbf{elem}(\texttt{m\_Ack}(Pid, Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\rightarrow (Pid \in \texttt{pids})$
$)$
$)$

Proof needs: [*stoller_inv9*]

stoller_inv15

Basic invariant, If an Ack-message from a process (Pid2) exists, then Pid2 is in the set *pids*.

$\forall Pid, Pid2.($
$(\mathbf{elem}(\texttt{m\_Ack}(Pid, Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\rightarrow (Pid2 \in \texttt{pids})$
$)$
$)$

Proof needs: []

stoller_inv16

Basic invariant, If a process (Pid) is alive and in state elec_1, no Ack-message from Pid exists.

$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_1}))$
$\rightarrow \neg\mathbf{elem}(\texttt{m\_Ack}(Pid2, Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv15*]

stoller_inv17

Basic invariant, If a Down-message contains a pid, that pid is also in the set *pids*.

$\forall Pid, Pid2.($
$(\mathbf{elem}(\texttt{m\_Down}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\rightarrow (Pid \in \texttt{pids})$
$)$
$)$

Proof needs: []

stoller_inv18

Basic invariant, If a Down-message contains a pid, that pid is not in the set *alive*.

$\forall Pid, Pid2.($
$(\mathbf{elem}(\texttt{m\_Down}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\rightarrow (Pid \notin \texttt{alive})$
$)$
$)$

Proof needs: [*stoller_inv17*]

stoller_inv19

System invariant, given two pids from the same host, if the larger pid is not alive, neither is the smaller one. I.e. new pids are always larger.

$\forall Pid, Pid2.($
$(((Pid2 \notin \texttt{alive})$
$\wedge (Pid \leq Pid2)$
$\wedge (\texttt{host}(Pid) = \texttt{host}(Pid2)))$
$\rightarrow (Pid \notin \texttt{alive})$
$)$
$)$
Proof needs: [*stoller_inv2*]

stoller_inv20

Basic invariant, Down-messages are only sent to processes on other nodes.
$\forall Pid, Pid2.($
$(\textbf{elem}(\texttt{m\_Down}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\rightarrow (\texttt{host}(Pid) \neq \texttt{host}(Pid2))$
$)$
$)$
Proof needs: [*stoller_inv2*]

stoller_inv21

Basic invariant, If a process (Pid) is alive and in state elec_2, then no Ack-message containing Pid as the acknowledging part exists.
$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow \neg\textbf{elem}(\texttt{m\_Ack}(Pid2, Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$
Proof needs: [*stoller_inv2*, *stoller_inv15*, *stoller_inv16*]

stoller_inv22

If there are two processes alive, and both are in state elec_2,no Ack-message from one of the processes to the other process exists.
$\forall Pid, Pid2, Pid3.($
$(((Pid \in \texttt{alive})$
$\wedge (Pid3 \in \texttt{alive})$
$\wedge (\texttt{host}(Pid) = \texttt{host}(Pid2))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2})$
$\wedge (\texttt{status}[\texttt{host}(Pid3)] = \texttt{elec\_2}))$
$\rightarrow \neg\textbf{elem}(\texttt{m\_Ack}(Pid3, Pid2), \texttt{queue}(\texttt{host}(Pid3)))$
$)$
$)$
Proof needs: [*stoller_inv2*, *stoller_inv8*, *stoller_inv12*, *stoller_inv14*, *stoller_inv59*]

stoller_inv23

If there are two processes alive (Pid and Pid3), both in state elec_2, no Down-message from host(Pid) to host(Pid3) exists. (Pid > Pid3).

$\forall Pid, Pid2, Pid3.($
$((\texttt{host}(Pid) > \texttt{host}(Pid3))$
$\wedge (Pid \in \texttt{alive})$
$\wedge (Pid3 \in \texttt{alive})$
$\wedge (\texttt{host}(Pid) = \texttt{host}(Pid2))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow \neg\mathbf{elem}(\texttt{m\_Down}(Pid2), \texttt{queue}(\texttt{host}(Pid3)))$
$)$
$)$
Proof needs: [*stoller_inv2*, *stoller_inv17*, *stoller_inv26*, *stoller_inv27*, *stoller_inv60*]

stoller_inv24

The highest prioritized process does not receive Halt-messages.
$\forall Pid, Pid2.($
$((\texttt{host}(Pid) = \texttt{s}(\texttt{zero}))$
$\rightarrow \neg\mathbf{elem}(\texttt{m\_Halt}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$)$
$)$
Proof needs: []

stoller_inv25

The highest prioritized process does not send Ack-messages.
$\forall Pid, Pid2.($
$((\texttt{host}(Pid) = \texttt{s}(\texttt{zero}))$
$\rightarrow \neg\mathbf{elem}(\texttt{m\_Ack}(Pid2, Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$
Proof needs: [*stoller_inv24*]

stoller_inv26

If there are two processes alive (Pid2 and Pid4) then there can not be a Down-message from BOTH host(Pid2) and host(Pid4) to host(Pid4) and host(Pid2) respectively.
$\forall Pid, Pid2, Pid3, Pid4.($
$((\texttt{host}(Pid2) \neq \texttt{host}(Pid4))$
$\wedge (Pid2 \in \texttt{alive})$
$\wedge (Pid4 \in \texttt{alive})$
$\wedge (\texttt{host}(Pid) = \texttt{host}(Pid2))$
$\wedge (\texttt{host}(Pid3) = \texttt{host}(Pid4)))$
$\rightarrow \neg(\mathbf{elem}(\texttt{m\_Down}(Pid), \texttt{queue}(\texttt{host}(Pid4)))$
$\wedge\ \mathbf{elem}(\texttt{m\_Down}(Pid3), \texttt{queue}(\texttt{host}(Pid2))))$
$)$
$)$
Proof needs: [*stoller_inv2*]

stoller_inv27

If there are two processes alive (Pid2 and Pid4), then it can not be the case that host(Pid4) is in the acks-set of Pid2 at the same time as there is a Down-message

enroute from host(Pid2) to host(Pid4).
$\forall Pid, Pid2, Pid3, Pid4.($
$(((\texttt{host}(Pid2) \neq \texttt{host}(Pid4))$
$\wedge\ (Pid2 \in \texttt{alive})$
$\wedge\ (Pid4 \in \texttt{alive})$
$\wedge\ (\texttt{host}(Pid) = \texttt{host}(Pid2))$
$\wedge\ (\texttt{host}(Pid3) = \texttt{host}(Pid4)))$
$\rightarrow \neg(\textbf{elem}(\texttt{m\_Down}(Pid), \texttt{queue}(\texttt{host}(Pid4)))$
$\wedge\ (\texttt{host}(Pid3) \in \texttt{down}[\texttt{host}(Pid2)]))$
$)$
$)$
Proof needs: [*stoller_inv2*, *stoller_inv26*]

stoller_inv28

If a process (Pid) is alive, then no Down-message containing Pid exists.
$\forall Pid, Pid2.($
$((Pid \in \texttt{alive})$
$\rightarrow \neg\textbf{elem}(\texttt{m\_Down}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$
Proof needs: [*stoller_inv17*]

stoller_inv30

If a process (Pid) is in state elec_2 and is waiting for the last Ack-message to complete the election, and this Ack-message is actually in Pid's queue, then no other process (Pid3) is currently the leader.
$\forall Pid, Pid2, Pid3.($
$(((Pid \in \texttt{alive})$
$\wedge\ \textbf{elem}(\texttt{m\_Ack}(Pid, Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge\ (\texttt{nbr\_proc} \leq \texttt{pendack}[\texttt{host}(Pid)])$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2})$
$\wedge\ (\texttt{host}(Pid2) = \texttt{pendack}[\texttt{host}(Pid)]))$
$\rightarrow \neg((Pid3 \in \texttt{alive})$
$\wedge\ (\texttt{status}[\texttt{host}(Pid3)] = \texttt{norm})$
$\wedge\ (\texttt{ldr}[\texttt{host}(Pid3)] = \texttt{host}(Pid3)))$
$)$
$)$
Proof needs: [*stoller_inv2*, *stoller_inv5*, *stoller_inv12*, *stoller_inv13*, *stoller_inv14*, *stoller_inv31*, *stoller_inv32*, *stoller_inv33*, *stoller_inv51*]

stoller_inv31

If Pid is in state elec_2 and have a pendack value greater than host(Pid3) and there is a Halt-message from Pid to some process, then no other process (Pid3) is a leader.

$\forall Pid, Pid2, Pid3.($
$(((\texttt{pendack}[\texttt{host}(Pid)] > \texttt{host}(Pid3))$
$\wedge (Pid \in \texttt{alive})$
$\wedge \mathbf{elem}(\texttt{m\_Halt}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow \neg((Pid3 \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid3)] = \texttt{norm})$
$\wedge (\texttt{ldr}[\texttt{host}(Pid3)] = \texttt{host}(Pid3)))$
$)$
$)$

Proof needs: [*stoller_inv5*, *stoller_inv6*, *stoller_inv7*, *stoller_inv8*, *stoller_inv36*, *stoller_inv38*, *stoller_inv39*, *stoller_inv40*, *stoller_inv53*, *stoller_inv57*, *stoller_inv1*]

stoller_inv32

If a process (Pid2) has sent an Ack-message to another process (Pid) and a Down-message from host(Pid) is in Pid2's queue, then Pid is not alive.

$\forall Pid, Pid2, Pid3.($
$((\mathbf{elem}(\texttt{m\_Down}(Pid3), \texttt{queue}(\texttt{host}(Pid2)))$
$\wedge \mathbf{elem}(\texttt{m\_Ack}(Pid, Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge (\texttt{host}(Pid) = \texttt{host}(Pid3)))$
$\rightarrow (Pid \notin \texttt{alive})$
$)$
$)$

Proof needs: [*stoller_inv8*, *stoller_inv18*, *stoller_inv74*, *stoller_inv75*]

stoller_inv33

If two processes are alive (Pid and Pid2) and there is a Ack-message from host(Pid2) in Pid's queue, then host(Pid) is not in the down-set of Pid2.

$\forall Pid, Pid2, Pid3.($
$(((Pid \in \texttt{alive})$
$\wedge (Pid2 \in \texttt{alive})$
$\wedge \mathbf{elem}(\texttt{m\_Ack}(Pid, Pid3), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge (\texttt{host}(Pid2) = \texttt{host}(Pid3)))$
$\rightarrow (\texttt{host}(Pid) \notin \texttt{down}[\texttt{host}(Pid2)])$
$)$
$)$

Proof needs: [*stoller_inv14*, *stoller_inv32*]

stoller_inv36

If a process is alive and has received an Ack-message from a process on another host, then no process on that host is currently a leader.

$\forall Pid, Pid2, Pid3.($
$(((Pid \in \texttt{alive})$
$\wedge\ \mathbf{elem}(\texttt{m\_Ack}(Pid, Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge\ (\texttt{host}(Pid2) = \texttt{host}(Pid3)))$
$\rightarrow \neg((Pid3 \in \texttt{alive})$
$\wedge\ (\texttt{status}[\texttt{host}(Pid3)] = \texttt{norm})$
$\wedge\ (\texttt{ldr}[\texttt{host}(Pid3)] = \texttt{host}(Pid3)))$
$)$
$)$
Proof needs: [*stoller_inv5*, *stoller_inv8*, *stoller_inv14*, *stoller_inv32*, *stoller_inv33*, *stoller_inv45*]

stoller_inv38

If a process is alive and has received a Down-message from a process on another host, then no process on that host is currently a leader.
$\forall Pid, Pid2, Pid3.($
$(((Pid \in \texttt{alive})$
$\wedge\ \mathbf{elem}(\texttt{m\_Down}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge\ (\texttt{host}(Pid2) = \texttt{host}(Pid3)))$
$\rightarrow \neg((Pid3 \in \texttt{alive})$
$\wedge\ (\texttt{status}[\texttt{host}(Pid3)] = \texttt{norm})$
$\wedge\ (\texttt{ldr}[\texttt{host}(Pid3)] = \texttt{host}(Pid3)))$
$)$
$)$
Proof needs: [*stoller_inv2*, *stoller_inv5*, *stoller_inv6*, *stoller_inv20*, *stoller_inv26*, *stoller_inv27*, *stoller_inv32*, *stoller_inv55*, *stoller_inv58a*]

stoller_inv39

The pendack value for a process is always $\leq$ nbr_proc.
$\forall Pid.((\texttt{pendack}[\texttt{host}(Pid)] \leq \texttt{nbr\_proc}))$
Proof needs: []

stoller_inv40

If a process is alive and has received a Down-message that makes lesser(Pid) a subset of down(Pid), then no process at a higher prioritized node is currently a leader.
$\forall Pid, Pid2, Pid3.($
$(((\texttt{host}(Pid) > \texttt{host}(Pid3))$
$\wedge\ (Pid \in \texttt{alive})$
$\wedge\ \mathbf{elem}(\texttt{m\_Down}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge\ (\texttt{lesser}(\texttt{host}(Pid)) \subseteq$
$(\texttt{down}[\texttt{host}(Pid)] \cup \{\texttt{host}(Pid2)\}))$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_1}))$
$\rightarrow \neg((Pid3 \in \texttt{alive})$
$\wedge\ (\texttt{status}[\texttt{host}(Pid3)] = \texttt{norm})$
$\wedge\ (\texttt{ldr}[\texttt{host}(Pid3)] = \texttt{host}(Pid3)))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv5*, *stoller_inv6*, *stoller_inv20*, *stoller_inv26*, *stoller_inv27*, *stoller_inv32*, *stoller_inv33*, *stoller_inv58a*, *stoller_inv58b*, *stoller_inv66*, *stoller_inv67*]

stoller_inv41

If a process (Pid) is alive and in state elec_2 and has received a Down-message from a process (Pid2) at a node with lower priority, then no process on that node is alive and in state elec_2.

$\forall Pid, Pid2, Pid3.($
$(((\texttt{host}(Pid2) > \texttt{host}(Pid))$
$\wedge (Pid \in \texttt{alive})$
$\wedge \textbf{elem}(\texttt{m\_Down}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge (\texttt{host}(Pid2) = \texttt{host}(Pid3))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow \neg((Pid3 \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid3)] = \texttt{elec\_2}))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv3*, *stoller_inv26*, *stoller_inv27*, *stoller_inv58a*]

stoller_inv42

If a process is the leader, then its election id is exactly the pid.

$\forall Pid.($
$(((Pid \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{norm})$
$\wedge (\texttt{ldr}[\texttt{host}(Pid)] = \texttt{host}(Pid)))$
$\rightarrow (\texttt{elid}[\texttt{host}(Pid)] = Pid)$
$)$
$)$

Proof needs: [*stoller_inv5*, *stoller_inv6*]

stoller_inv43

NormQ-messages are only sent to processes with lower priority.

$\forall Pid, Pid2.($
$(\textbf{elem}(\texttt{m\_NormQ}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\rightarrow (\texttt{host}(Pid2) > \texttt{host}(Pid))$
$)$
$)$

Proof needs: [*stoller_inv42*]

stoller_inv44

A process at the lowest prioritized host does never receive a NotNorm-message.

$\forall Pid, Pid2.($
$((\texttt{host}(Pid) = \texttt{nbr\_proc})$
$\rightarrow \neg\textbf{elem}(\texttt{m\_NotNorm}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$)$
$)$

Proof needs: [*stoller_inv43*]

stoller_inv45

If a process (Pid) is alive and in state elec_2, then no process at a host with higherpriority has node(Pid) in its down-set.

$\forall Pid, Pid2.($
$(((\texttt{host}(Pid) > \texttt{host}(Pid2))$
$\wedge (Pid \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow (\texttt{host}(Pid2) \in \texttt{down}[\texttt{host}(Pid)])$
$)$
$)$

Proof needs: []

stoller_inv50

If a process (Pid) is alive, in state elec_2, waiting for a reply from the host with lowest priority and have a Down-reply in its queue, then no other process is a leader.

$\forall Pid, Pid2, Pid3.($
$(((Pid \in \texttt{alive})$
$\wedge \mathbf{elem}(\texttt{m\_Down}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge (\texttt{nbr\_proc} \leq \texttt{pendack}[\texttt{host}(Pid)])$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2})$
$\wedge (\texttt{host}(Pid2) = \texttt{pendack}[\texttt{host}(Pid)]))$
$\rightarrow \neg((Pid3 \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid3)] = \texttt{norm})$
$\wedge (\texttt{ldr}[\texttt{host}(Pid3)] = \texttt{host}(Pid3)))$
$)$
$)$

Proof needs: [*stoller_inv1*, *stoller_inv5*, *stoller_inv6*, *stoller_inv14*, *stoller_inv51*, *stoller_inv52*, *stoller_inv53*, *stoller_inv54*, *stoller_inv55*, *stoller_inv56*]

stoller_inv51

If two processes (Pid and Pid2) are alive and in state elec_2, and host(Pid) < host(Pid2), then the pendack value of Pid2 is larger than the pendack value of Pid.

$\forall Pid, Pid2.($
$(((\texttt{host}(Pid2) > \texttt{host}(Pid))$
$\wedge (Pid \in \texttt{alive})$
$\wedge (Pid2 \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2})$
$\wedge (\texttt{status}[\texttt{host}(Pid2)] = \texttt{elec\_2}))$
$\rightarrow (\texttt{pendack}[\texttt{host}(Pid2)] > \texttt{pendack}[\texttt{host}(Pid)])$
$)$
$)$

Proof needs: [*stoller_inv3*, *stoller_inv6*, *stoller_inv22*, *stoller_inv57*, *stoller_inv58a*, *stoller_inv58b*]

stoller_inv52

If a process (Pid) is alive, in state elec_2 and waiting for an Ack-message from the second last process, then if this message is in its queue and there is also a Down-message from the last process in the queue (i.e. Pid can become leader without further communication) then no other process is currently a leader.

$$
\begin{array}{l}
\forall \mathit{Pid}, \mathit{Pid2}, \mathit{Pid3}, \mathit{Pid4}.(\\
\quad (((\mathit{Pid} \in \texttt{alive}) \\
\qquad \wedge\ \mathbf{elem}(\texttt{m\_Down}(\mathit{Pid2}), \texttt{queue}(\texttt{host}(\mathit{Pid}))) \\
\qquad \wedge\ \mathbf{elem}(\texttt{m\_Ack}(\mathit{Pid}, \mathit{Pid3}), \texttt{queue}(\texttt{host}(\mathit{Pid}))) \\
\qquad \wedge\ (\texttt{nbr\_proc} \leq \texttt{s}(\texttt{pendack}[\texttt{host}(\mathit{Pid})])) \\
\qquad \wedge\ (\texttt{status}[\texttt{host}(\mathit{Pid})] = \texttt{elec\_2}) \\
\qquad \wedge\ (\texttt{host}(\mathit{Pid3}) = \texttt{pendack}[\texttt{host}(\mathit{Pid})]) \\
\qquad \wedge\ (\texttt{host}(\mathit{Pid2}) = \texttt{s}(\texttt{pendack}[\texttt{host}(\mathit{Pid})])) \\
\qquad \rightarrow \neg((\mathit{Pid4} \in \texttt{alive}) \\
\qquad\quad \wedge\ (\texttt{status}[\texttt{host}(\mathit{Pid4})] = \texttt{norm}) \\
\qquad\quad \wedge\ (\texttt{ldr}[\texttt{host}(\mathit{Pid4})] = \texttt{host}(\mathit{Pid4}))) \\
\quad) \\
)
\end{array}
$$

Proof needs: [*stoller_inv2*, *stoller_inv5*, *stoller_inv6*, *stoller_inv12*, *stoller_inv14*, *stoller_inv20*, *stoller_inv22*, *stoller_inv13*, *stoller_inv31*, *stoller_inv36*, *stoller_inv38*, *stoller_inv53*, *stoller_inv55*, *stoller_inv57*]

stoller_inv53

If a process (Pid) is alive and in state elec_2, then no process (Pid2) with host(Pid2) < pendack(Pid) is currently a leader.

$$
\begin{array}{l}
\forall \mathit{Pid}, \mathit{Pid2}.(\\
\quad (((\texttt{pendack}[\texttt{host}(\mathit{Pid})] > \texttt{host}(\mathit{Pid2})) \\
\qquad \wedge\ (\mathit{Pid} \in \texttt{alive}) \\
\qquad \wedge\ (\texttt{status}[\texttt{host}(\mathit{Pid})] = \texttt{elec\_2})) \\
\qquad \rightarrow \neg((\mathit{Pid2} \in \texttt{alive}) \\
\qquad\quad \wedge\ (\texttt{status}[\texttt{host}(\mathit{Pid2})] = \texttt{norm}) \\
\qquad\quad \wedge\ (\texttt{ldr}[\texttt{host}(\mathit{Pid2})] = \texttt{host}(\mathit{Pid2}))) \\
\quad) \\
)
\end{array}
$$

Proof needs: [*stoller_inv5*, *stoller_inv6*, *stoller_inv22*, *stoller_inv36*, *stoller_inv57*, *stoller_inv58a*, *stoller_inv58b*, *stoller_inv66*, *stoller_inv67*]

stoller_inv54

If a process (Pid) is alive, in state elec_2 and has received a Down-message from a process on another host, then no process on that host is currently a leader.

$\forall Pid, Pid2, Pid3.($
$(((Pid \in \texttt{alive})$
$\wedge\ \mathbf{elem}(\texttt{m\_Down}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge\ (\texttt{host}(Pid2) = \texttt{host}(Pid3))$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow \neg((Pid3 \in \texttt{alive})$
$\wedge\ (\texttt{status}[\texttt{host}(Pid3)] = \texttt{norm})$
$\wedge\ (\texttt{ldr}[\texttt{host}(Pid3)] = \texttt{host}(Pid3)))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv5*, *stoller_inv6*, *stoller_inv32*, *stoller_inv38*, *stoller_inv41*, *stoller_inv55*, *stoller_inv57*]

stoller_inv55

If a process (Pid) on the host with lowest priority is alive, in state elec_1 and have a Down-message in its queue that makes the down-set complete, then no other process is alive that has a Down-message from host(Pid).

$\forall Pid, Pid2, Pid3, Pid4.($
$(((Pid \in \texttt{alive})$
$\wedge\ \mathbf{elem}(\texttt{m\_Down}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge\ (\texttt{lesser}(\texttt{host}(Pid)) \subseteq$
$(\texttt{down}[\texttt{host}(Pid)] \cup \{\texttt{host}(Pid2)\}))$
$\wedge\ (\texttt{host}(Pid) = \texttt{nbr\_proc})$
$\wedge\ (\texttt{host}(Pid) = \texttt{host}(Pid3))$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_1}))$
$\rightarrow \neg((Pid4 \in \texttt{alive})$
$\wedge\ \mathbf{elem}(\texttt{m\_Down}(Pid3), \texttt{queue}(\texttt{host}(Pid4))))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv5*, *stoller_inv20*, *stoller_inv26*, *stoller_inv27*, *stoller_inv40*, *stoller_inv44*, *stoller_inv61*]

stoller_inv56

A lot like 52, but here there are two Down-messages instead of one Down and one Ack i.e. a process is waiting for a message from the second last process and have this message in its queue. That is Pid can become leader without further communication.

$\forall Pid, Pid2, Pid3, Pid4.($
$(((Pid \in \texttt{alive})$
$\wedge (\texttt{nbr\_proc} \leq \texttt{s}(\texttt{host}(Pid)))$
$\wedge \mathbf{elem}(\texttt{m\_Down}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge \mathbf{elem}(\texttt{m\_Down}(Pid3), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge (\texttt{lesser}(\texttt{host}(Pid)) \subseteq$
$(\texttt{down}[\texttt{host}(Pid)] \cup \{\texttt{host}(Pid2)\}))$
$\wedge (\texttt{host}(Pid3) = \texttt{s}(\texttt{host}(Pid)))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_1}))$
$\rightarrow \neg((Pid4 \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid4)] = \texttt{norm})$
$\wedge (\texttt{ldr}[\texttt{host}(Pid4)] = \texttt{host}(Pid4)))$
$)$
$)$

Proof needs: [*stoller_inv1*, *stoller_inv2*, *stoller_inv5*, *stoller_inv6*, *stoller_inv14*, *stoller_inv20*, *stoller_inv26*, *stoller_inv27*, *stoller_inv58a*, *stoller_inv58b*, *stoller_inv59*, *stoller_inv63*, *stoller_inv64*, *stoller_inv65*, *stoller_inv66*, *stoller_inv67*]

stoller_inv57

If two processes (Pid and Pid2) are alive and in state elec_2, then the pendack value of the one highest priority (Pid) is less than or equal to host(Pid2)

$\forall Pid, Pid2.($
$(((\texttt{host}(Pid2) > \texttt{host}(Pid))$
$\wedge (Pid \in \texttt{alive})$
$\wedge (Pid2 \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2})$
$\wedge (\texttt{status}[\texttt{host}(Pid2)] = \texttt{elec\_2}))$
$\rightarrow (\texttt{pendack}[\texttt{host}(Pid)] \leq \texttt{host}(Pid2))$
$)$
$)$

Proof needs: [*stoller_inv6*, *stoller_inv22*, *stoller_inv23*, *stoller_inv58a*, *stoller_inv58b*]

stoller_inv58a

If Pid and Pid3 are alive, Pid is in state elec_2 and there is a Down-message from host(Pid) in Pid3's queue, then the pendack value of Pid is less than or equal to host(Pid3).

$\forall Pid, Pid2, Pid3.($
$(((Pid \in \texttt{alive})$
$\wedge (Pid3 \in \texttt{alive})$
$\wedge \mathbf{elem}(\texttt{m\_Down}(Pid2), \texttt{queue}(\texttt{host}(Pid3)))$
$\wedge (\texttt{host}(Pid) = \texttt{host}(Pid2))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow (\texttt{pendack}[\texttt{host}(Pid)] \leq \texttt{host}(Pid3))$
$)$
$)$

Proof needs: [*stoller_inv6*, *stoller_inv20*, *stoller_inv26*, *stoller_inv27*, *stoller_inv32*]

stoller_inv58b

If Pid and Pid2 are alive, Pid is in state elec_2 and host(Pid) is in the down-set of Pid2, then the pendack value of Pid is less than or equal to host(Pid2).

$$
\begin{array}{l}
\forall \mathit{Pid}, \mathit{Pid2}.(\\
\quad (((\mathit{Pid} \in \texttt{alive}) \\
\qquad \wedge (\mathit{Pid2} \in \texttt{alive}) \\
\qquad \wedge (\texttt{host}(\mathit{Pid}) \in \texttt{down}[\texttt{host}(\mathit{Pid2})]) \\
\qquad \wedge (\texttt{status}[\texttt{host}(\mathit{Pid})] = \texttt{elec\_2})) \\
\qquad \rightarrow (\texttt{pendack}[\texttt{host}(\mathit{Pid})] \leq \texttt{host}(\mathit{Pid2})) \\
\quad) \\
)
\end{array}
$$

Proof needs: [*stoller_inv6*, *stoller_inv20*, *stoller_inv27*, *stoller_inv33*, *stoller_inv58a*, *stoller_inv70*]

stoller_inv59

If two processes (Pid and Pid4) are alive, Pid is in state elec_1 and Pid4 is in state elec_2 and Pid has the last Down-message it needs to go to state elec_2 in its queue, then there is no Ack-message from host(Pid) in the queue of Pid4.

$$
\begin{array}{l}
\forall \mathit{Pid}, \mathit{Pid2}, \mathit{Pid3}, \mathit{Pid4}.(\\
\quad (((\mathit{Pid} \in \texttt{alive}) \\
\qquad \wedge (\mathit{Pid4} \in \texttt{alive}) \\
\qquad \wedge (\texttt{host}(\mathit{Pid4}) \leq \texttt{host}(\mathit{Pid})) \\
\qquad \wedge \mathbf{elem}(\texttt{m\_Down}(\mathit{Pid2}), \texttt{queue}(\texttt{host}(\mathit{Pid}))) \\
\qquad \wedge (\texttt{lesser}(\texttt{host}(\mathit{Pid})) \subseteq \\
\qquad\qquad (\texttt{down}[\texttt{host}(\mathit{Pid})] \cup \{\texttt{host}(\mathit{Pid2})\})) \\
\qquad \wedge (\texttt{host}(\mathit{Pid3}) = \texttt{host}(\mathit{Pid})) \\
\qquad \wedge (\texttt{status}[\texttt{host}(\mathit{Pid})] = \texttt{elec\_1}) \\
\qquad \wedge (\texttt{status}[\texttt{host}(\mathit{Pid4})] = \texttt{elec\_2})) \\
\qquad \rightarrow \neg\mathbf{elem}(\texttt{m\_Ack}(\mathit{Pid4}, \mathit{Pid3}), \texttt{queue}(\texttt{host}(\mathit{Pid4}))) \\
\quad) \\
)
\end{array}
$$

Proof needs: [*stoller_inv2*, *stoller_inv14*, *stoller_inv16*, *stoller_inv26*, *stoller_inv27*, *stoller_inv32*, *stoller_inv33*, *stoller_inv58a*]

stoller_inv60

If a process (Pid) is the leader, then no process (Pid2) with lower priority is alive and in state elec_2.

$$
\begin{array}{l}
\forall \mathit{Pid}, \mathit{Pid2}.(\\
\quad (((\texttt{host}(\mathit{Pid2}) > \texttt{host}(\mathit{Pid})) \\
\qquad \wedge (\mathit{Pid} \in \texttt{alive}) \\
\qquad \wedge (\texttt{status}[\texttt{host}(\mathit{Pid})] = \texttt{norm}) \\
\qquad \wedge (\texttt{ldr}[\texttt{host}(\mathit{Pid})] = \texttt{host}(\mathit{Pid}))) \\
\qquad \rightarrow \neg((\mathit{Pid2} \in \texttt{alive}) \\
\qquad\qquad \wedge (\texttt{status}[\texttt{host}(\mathit{Pid2})] = \texttt{elec\_2})) \\
\quad) \\
)
\end{array}
$$

Proof needs: [*stoller_inv5*, *stoller_inv6*, *stoller_inv22*, *stoller_inv23*, *stoller_inv40*, *stoller_inv57*]

stoller_inv61

If a process (Pid) have a leader (Pid2) then there are no Down-messages from host(Pid) in Pid2's queue.

$\forall Pid, Pid2, Pid3.($
$(((\texttt{host}(Pid) \neq \texttt{host}(Pid2))$
$\wedge (Pid \in \texttt{alive})$
$\wedge (\texttt{host}(Pid) = \texttt{host}(Pid3))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{norm})$
$\wedge (\texttt{ldr}[\texttt{host}(Pid)] = \texttt{host}(Pid2)))$
$\rightarrow \neg\textbf{elem}(\texttt{m\_Down}(Pid3), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv71*]

stoller_inv63

If a process (Pid) have a leader (Pid2) and there is a Down-messge in the queue of Pid from a process at host(Pid2) the Pid2 is no longer the leader.

$\forall Pid, Pid2, Pid3.($
$(((\texttt{host}(Pid) \neq \texttt{host}(Pid2))$
$\wedge (Pid \in \texttt{alive})$
$\wedge \textbf{elem}(\texttt{m\_Down}(Pid3), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge (\texttt{host}(Pid2) = \texttt{host}(Pid3))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{norm})$
$\wedge (\texttt{ldr}[\texttt{host}(Pid)] = \texttt{host}(Pid2)))$
$\rightarrow \neg((Pid2 \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid2)] = \texttt{norm})$
$\wedge (\texttt{ldr}[\texttt{host}(Pid2)] = \texttt{host}(Pid2)))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv5*, *stoller_inv6*, *stoller_inv27*, *stoller_inv32*, *stoller_inv61*, *stoller_inv64*, *stoller_inv87*, *stoller_inv88*]

stoller_inv64

If a process (Pid) is waiting for a Ldr-message from another process (Pid2) and there is a Down-message from host(Pid2) in Pid's queue, then Pid2 is not the leader.

$\forall Pid, Pid2, Pid3.($
$(((\texttt{host}(Pid) \neq \texttt{host}(Pid2))$
$\wedge (Pid \in \texttt{alive})$
$\wedge \textbf{elem}(\texttt{m\_Down}(Pid3), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge (\texttt{host}(Pid2) = \texttt{host}(Pid3))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{wait})$
$\wedge (\texttt{host}(\texttt{elid}[\texttt{host}(Pid)]) = \texttt{host}(Pid2)))$
$\rightarrow \neg((Pid2 \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid2)] = \texttt{norm})$
$\wedge (\texttt{ldr}[\texttt{host}(Pid2)] = \texttt{host}(Pid2)))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv5*, *stoller_inv6*, *stoller_inv26*, *stoller_inv27*, *stoller_inv32*, *stoller_inv58a*, *stoller_inv66*]

stoller_inv65

If a process (Pid) is alive and has a Down-message from a lower prioritized host in its queue, then no process on that host is currently the leader.

$\forall Pid, Pid2, Pid3.($
$(((\texttt{host}(Pid2) > \texttt{host}(Pid))$
$\wedge\ (Pid \in \texttt{alive})$
$\wedge\ \textbf{elem}(\texttt{m\_Down}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge\ (\texttt{host}(Pid2) = \texttt{host}(Pid3)))$
$\rightarrow \neg((Pid3 \in \texttt{alive})$
$\wedge\ (\texttt{status}[\texttt{host}(Pid3)] = \texttt{norm})$
$\wedge\ (\texttt{ldr}[\texttt{host}(Pid3)] = \texttt{host}(Pid3)))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv5*, *stoller_inv23*, *stoller_inv26*, *stoller_inv27*]

stoller_inv66

If a process (Pid) is the leader, then there is no other Process (Pid3) that is both alive and has a Down-message from a process on host(Pid).

$\forall Pid, Pid2, Pid3.($
$(((Pid \in \texttt{alive})$
$\wedge\ (\texttt{host}(Pid2) = \texttt{host}(Pid))$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{norm})$
$\wedge\ (\texttt{ldr}[\texttt{host}(Pid)] = \texttt{host}(Pid)))$
$\rightarrow \neg((Pid3 \in \texttt{alive})$
$\wedge\ \textbf{elem}(\texttt{m\_Down}(Pid2), \texttt{queue}(\texttt{host}(Pid3))))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv5*, *stoller_inv6*, *stoller_inv20*, *stoller_inv26*, *stoller_inv27*, *stoller_inv32*, *stoller_inv87*, *stoller_inv88*]

stoller_inv67

If a process (Pid) is the leader, then there is no other Process (Pid2) that is both alive and has host(Pid) in its down-set.

$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{norm})$
$\wedge\ (\texttt{ldr}[\texttt{host}(Pid)] = \texttt{host}(Pid)))$
$\rightarrow \neg((Pid2 \in \texttt{alive})$
$\wedge\ (\texttt{host}(Pid) \in \texttt{down}[\texttt{host}(Pid2)]))$
$)$
$)$

Proof needs: [*stoller_inv3*, *stoller_inv5*, *stoller_inv6*, *stoller_inv20*, *stoller_inv27*, *stoller_inv33*, *stoller_inv66*, *stoller_inv70*, *stoller_inv88*, *stoller_inv89*]

stoller_inv68

If a process is the leader and process (Pid2) with higher priority is alive, then Pid2 is not in state elec_2 with a pendac value greater than host(Pid).

$$
\begin{array}{l}
\forall Pid, Pid2.(\\
\quad (((\texttt{host}(Pid) > \texttt{host}(Pid2)) \\
\qquad \wedge (Pid \in \texttt{alive}) \\
\qquad \wedge (Pid2 \in \texttt{alive}) \\
\qquad \wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{norm}) \\
\qquad \wedge (\texttt{ldr}[\texttt{host}(Pid)] = \texttt{host}(Pid))) \\
\qquad\quad \rightarrow \neg((\texttt{pendack}[\texttt{host}(Pid2)] > \texttt{host}(Pid)) \\
\qquad\qquad \wedge (\texttt{status}[\texttt{host}(Pid2)] = \texttt{elec\_2})) \\
\quad) \\
)
\end{array}
$$

Proof needs: [*stoller_inv5*, *stoller_inv6*, *stoller_inv36*, *stoller_inv38*, *stoller_inv57*, *stoller_inv58a*, *stoller_inv58b*]

stoller_inv69

The down-set of a process (Pid) never contains host(Pid).

$$
\begin{array}{l}
\forall Pid.(\\
\quad (\texttt{host}(Pid) \notin \texttt{down}[\texttt{host}(Pid)]) \\
)
\end{array}
$$

Proof needs: [*stoller_inv20*]

stoller_inv70

If two processes (Pid and Pid2) are alive, then it is not the case that they both have each other in their down-sets.

$$
\begin{array}{l}
\forall Pid, Pid2.(\\
\quad (((Pid \in \texttt{alive}) \\
\qquad \wedge (Pid2 \in \texttt{alive}) \\
\qquad \wedge (\texttt{host}(Pid2) \in \texttt{down}[\texttt{host}(Pid)])) \\
\qquad\quad \rightarrow (\texttt{host}(Pid) \notin \texttt{down}[\texttt{host}(Pid2)]) \\
\quad) \\
)
\end{array}
$$

Proof needs: [*stoller_inv20*, *stoller_inv27*]

stoller_inv71

If a process (Pid) is waiting for another process (Pid2) and a Ldr-message from Pid2 is in Pid's queue, then Pid2 does not have a Down-message from host(Pid) in its queue.

$\forall Pid, Pid2, Pid3.($
$(((\texttt{host}(Pid) \neq \texttt{host}(Pid2))$
$\wedge (Pid \in \texttt{alive})$
$\wedge \mathbf{elem}(\texttt{m\_Ldr}(\texttt{elid}[\texttt{host}(Pid)]), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge (\texttt{host}(Pid) = \texttt{host}(Pid3))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{wait})$
$\wedge (\texttt{host}(\texttt{elid}[\texttt{host}(Pid)]) = \texttt{host}(Pid2)))$
$\rightarrow \neg\mathbf{elem}(\texttt{m\_Down}(Pid3), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv6*, *stoller_inv17*, *stoller_inv18*, *stoller_inv72*, *stoller_inv73*, *stoller_inv75*, *stoller_inv102*, *stoller_inv103*]

stoller_inv72

If a process (Pid) is alive and has a Halt-message from Pid2 in its queue, then no Ldr-message from host(Pid2) is in Pid's queue.

$\forall Pid, Pid2, Pid4.($
$(((Pid2 \leq Pid4)$
$\wedge (Pid \in \texttt{alive})$
$\wedge \mathbf{elem}(\texttt{m\_Halt}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge (\texttt{host}(Pid2) = \texttt{host}(Pid4)))$
$\rightarrow \neg\mathbf{elem}(\texttt{m\_Ldr}(Pid4), \texttt{queue}(\texttt{host}(Pid)))$
$)$
$)$

Proof needs: [*stoller_inv6*, *stoller_inv8*, *stoller_inv75*, *stoller_inv76*, *stoller_inv77*, *stoller_inv78*, *stoller_inv79*, *stoller_inv80*, *stoller_inv102*]

stoller_inv73

If a process (Pid) is waiting for another process (Pid2) and host(Pid) is in Pid2's ack-set, then Pid2 does not have a Down-message from host(Pid) in its queue.

$\forall Pid, Pid2, Pid3.($
$(((\texttt{host}(Pid) \neq \texttt{host}(Pid2))$
$\wedge (Pid \in \texttt{alive})$
$\wedge (\texttt{host}(Pid) \in \texttt{acks}[\texttt{host}(Pid2)])$
$\wedge (\texttt{host}(Pid) = \texttt{host}(Pid3))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{wait})$
$\wedge (\texttt{host}(\texttt{elid}[\texttt{host}(Pid)]) = \texttt{host}(Pid2)))$
$\rightarrow \neg\mathbf{elem}(\texttt{m\_Down}(Pid3), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv4*, *stoller_inv6*, *stoller_inv8*, *stoller_inv17*, *stoller_inv18*, *stoller_inv75*, *stoller_inv77*, *stoller_inv102*, *stoller_inv103*]

stoller_inv74

On a host, if a process (Pid) is alive, then no other process (Pid2) with a larger process id exists.

$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive}) \ \wedge \ (\texttt{host}(Pid) = \texttt{host}(Pid2)))$
$\rightarrow (Pid2 \leq Pid)$
$)$
$)$
Proof needs: []

stoller_inv75

The message queue for a process is always ordered.
$\forall Pid.(\textbf{ordered}(\texttt{queue}(\texttt{host}(Pid))))$
Proof needs: [*stoller_inv74*]

stoller_inv76

If a Ldr-message contains Pid, then Pid is in the pids-set.
$\forall Pid, Pid2.($
$(\textbf{elem}(\texttt{m\_Ldr}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\rightarrow (Pid \in \texttt{pids})$
$)$
$)$
Proof needs: [*stoller_inv81*]

stoller_inv77

If a Halt-message containing Pid is in the queue of Pid2, then host(Pid2) is not in the acks-set of Pid.
$\forall Pid, Pid2.($
$(\textbf{elem}(\texttt{m\_Halt}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\rightarrow (\texttt{host}(Pid2) \notin \texttt{acks}[\texttt{host}(Pid)])$
$)$
$)$
Proof needs: [*stoller_inv6*, *stoller_inv9*, *stoller_inv78*, *stoller_inv82*, *stoller_inv102*]

stoller_inv78

If a Halt-message containing Pid is in the queue of Pid2, then there is no Ack-message containing Pid2 in Pid2's queue from an earlier incarnation of Pid.
$\forall Pid, Pid2, Pid3.($
$(((Pid \leq Pid3)$
$\wedge \textbf{elem}(\texttt{m\_Halt}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\wedge (\texttt{host}(Pid) = \texttt{host}(Pid3)))$
$\rightarrow \neg\textbf{elem}(\texttt{m\_Ack}(Pid3, Pid2), \texttt{queue}(\texttt{host}(Pid3)))$
$)$
$)$
Proof needs: [*stoller_inv7*, *stoller_inv8*, *stoller_inv12*, *stoller_inv13*, *stoller_inv14*, *stoller_inv75*, *stoller_inv83*, *stoller_inv102*]

stoller_inv79

If a process (Pid) is alive and in state elec_2, then no Ldr-message containing Pid exists.
$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow \neg \textbf{elem}(\texttt{m\_Ldr}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$
Proof needs: [*stoller_inv42*, *stoller_inv76*, *stoller_inv80*]

stoller_inv80

If a process (Pid) is alive and in state elec_1, then no Ldr-message containing Pid exists.
$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_1}))$
$\rightarrow \neg \textbf{elem}(\texttt{m\_Ldr}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$
Proof needs: [*stoller_inv2*, *stoller_inv42*, *stoller_inv76*]

stoller_inv81

If a process (Pid) is alive and in state elec_2, then Pid is in the pids-set.
$\forall Pid.($
$(((Pid \in \texttt{alive})$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow (Pid \in \texttt{pids})$
$)$
$)$
Proof needs: [*stoller_inv2*]

stoller_inv82

If a process (Pid) is alive and in state elec_2 and the pendack value of Pid is less than or equal to host(Pid2), then host(Pid2) is not in Pid's acks-set.
$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge (\texttt{pendack}[\texttt{host}(Pid)] \leq \texttt{host}(Pid2))$
$\wedge (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow (\texttt{host}(Pid2) \notin \texttt{acks}[\texttt{host}(Pid)])$
$)$
$)$
Proof needs: []

stoller_inv83

There are never two Halt-messages with the same Pid in a queue.
$\forall Pid, Pid2.($
$\quad \neg \mathbf{twice}(\mathtt{queue}(\mathtt{host}(Pid)), \mathtt{m\_Halt}(Pid2))$
$)$
Proof needs: [*stoller_inv9*, *stoller_inv10*, *stoller_inv84*]

stoller_inv84

If a process (Pid) is alive, in state elec_2 and has sent a Halt-message to Pid2, then host(Pid2) is less than or equal to the pendack value of Pid.
$\forall Pid, Pid2.($
$\quad (((Pid \in \mathtt{alive})$
$\qquad \wedge \mathbf{elem}(\mathtt{m\_Halt}(Pid), \mathtt{queue}(\mathtt{host}(Pid2)))$
$\qquad \wedge (\mathtt{status}[\mathtt{host}(Pid)] = \mathtt{elec\_2}))$
$\qquad \rightarrow (\mathtt{host}(Pid2) \leq \mathtt{pendack}[\mathtt{host}(Pid)])$
$\quad)$
$)$
Proof needs: [*stoller_inv2*, *stoller_inv9*, *stoller_inv10*]

stoller_inv87

If a process (Pid) is alive, in state elec_2 and has host(Pid3) in its acks-set, then no Down-message from host(Pid) is in Pid3's queue.
$\forall Pid, Pid2, Pid3.($
$\quad (((Pid \in \mathtt{alive})$
$\qquad \wedge (\mathtt{host}(Pid3) \in \mathtt{acks}[\mathtt{host}(Pid)])$
$\qquad \wedge (\mathtt{host}(Pid) = \mathtt{host}(Pid2))$
$\qquad \wedge (\mathtt{status}[\mathtt{host}(Pid)] = \mathtt{elec\_2}))$
$\qquad \rightarrow \neg \mathbf{elem}(\mathtt{m\_Down}(Pid2), \mathtt{queue}(\mathtt{host}(Pid3)))$
$\quad)$
$)$
Proof needs: [*stoller_inv6*, *stoller_inv32*]

stoller_inv88

If a process (Pid) is alive and in state elec_2, then lesser(Pid) minus host(Pid) is a subset of the union of Pid's down-set and acks-set.
$\forall Pid.($
$\quad (((Pid \in \mathtt{alive})$
$\qquad \wedge (\mathtt{status}[\mathtt{host}(Pid)] = \mathtt{elec\_2}))$
$\qquad \rightarrow ((\mathtt{lesser}(\mathtt{pendack}[\mathtt{host}(Pid)]) - \{\mathtt{host}(Pid)\}) \subseteq$
$\qquad\qquad (\mathtt{down}[\mathtt{host}(Pid)] \cup \mathtt{acks}[\mathtt{host}(Pid)]))$
$\quad)$
$)$
Proof needs: []

stoller_inv89

If a process (Pid) is alive, in state elec_2, has host(Pid2) in its acks-set and Pid2 is alive, then host(Pid) is not in Pid2's down-set.

$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge\ (Pid2 \in \texttt{alive})$
$\wedge\ (\texttt{host}(Pid2) \in \texttt{acks}[\texttt{host}(Pid)])$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow\ (\texttt{host}(Pid) \notin \texttt{down}[\texttt{host}(Pid2)])$
$)$
$)$

Proof needs: [*stoller_inv6*, *stoller_inv33*, *stoller_inv87*]

stoller_inv90

If a process (Pid) is waiting for another process (Pid2) and a Ldr-message from Pid2 is in Pid's queue, then there is no Ack-message from host(Pid) in Pid2's queue.

$\forall Pid, Pid2, Pid3.($
$(((Pid \in \texttt{alive})$
$\wedge\ \textbf{elem}(\texttt{m\_Ldr}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge\ (\texttt{host}(Pid) = \texttt{host}(Pid3))$
$\wedge\ (\texttt{elid}[\texttt{host}(Pid)] = Pid2)$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{wait}))$
$\rightarrow\ \neg\textbf{elem}(\texttt{m\_Ack}(Pid2, Pid3), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$

Proof needs: [*stoller_inv6*, *stoller_inv72*, *stoller_inv93*, *stoller_inv94*, *stoller_inv103*]

stoller_inv93

No queue contains two identical Ack-messages.

$\forall Pid, Pid2.($
$\neg\textbf{twice}(\texttt{queue}(\texttt{host}(Pid)), \texttt{m\_Ack}(Pid, Pid2))$
$)$

Proof needs: [*stoller_inv96*]

stoller_inv94

If a process (Pid) is alive, in state elec_2 and has host(Pid2) in its acks-set, then no Ack-message from Pid2 is in Pid's queue.

$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge\ (\texttt{host}(Pid2) \in \texttt{acks}[\texttt{host}(Pid)])$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow\ \neg\textbf{elem}(\texttt{m\_Ack}(Pid, Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$)$
$)$

Proof needs: [*stoller_inv6*, *stoller_inv8*, *stoller_inv93*, *stoller_inv97*, *stoller_inv104*]

stoller_inv95

If a process (Pid) is waiting for another process (Pid2) and a Ldr-message from Pid2 is in Pid's queue, then there is no Down-message from host(Pid) in Pid2's queue.

$\forall Pid, Pid2, Pid3.($
$(((Pid \in \texttt{alive})$
$\wedge\ \mathbf{elem}(\texttt{m\_Ldr}(Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge\ (\texttt{host}(Pid) = \texttt{host}(Pid3))$
$\wedge\ (\texttt{elid}[\texttt{host}(Pid)] = Pid2)$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{wait}))$
$\rightarrow \neg\mathbf{elem}(\texttt{m\_Down}(Pid3), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv6*, *stoller_inv17*, *stoller_inv18*, *stoller_inv72*, *stoller_inv73*, *stoller_inv75*, *stoller_inv102*, *stoller_inv103*]

stoller_inv96

Between a pair of processes, there is never both an Ack-message and a Halt-message in the respective queues.

$\forall Pid, Pid2.($
$\neg(\mathbf{elem}(\texttt{m\_Halt}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$\wedge\ \mathbf{elem}(\texttt{m\_Ack}(Pid, Pid2), \texttt{queue}(\texttt{host}(Pid))))$
$)$

Proof needs: [*stoller_inv8*, *stoller_inv12*, *stoller_inv13*, *stoller_inv14*, *stoller_inv83*]

stoller_inv97

If a process (Pid) is alive, in state elec_2 and has host(Pid2) in its acks-set, then no Halt-message from Pid is in Pid2's queue.

$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge\ (\texttt{host}(Pid2) \in \texttt{acks}[\texttt{host}(Pid)])$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow \neg\mathbf{elem}(\texttt{m\_Halt}(Pid), \texttt{queue}(\texttt{host}(Pid2)))$
$)$
$)$

Proof needs: [*stoller_inv6*, *stoller_inv96*, *stoller_inv98*]

stoller_inv98

If a process (Pid) is alive, in state elec_2 and has host(Pid2) in its acks-set, then host(Pid2) is less than or equal to Pid's pendack value.

$\forall Pid, Pid2.($
$(((Pid \in \texttt{alive})$
$\wedge\ (\texttt{host}(Pid2) \in \texttt{acks}[\texttt{host}(Pid)])$
$\wedge\ (\texttt{status}[\texttt{host}(Pid)] = \texttt{elec\_2}))$
$\rightarrow (\texttt{host}(Pid2) \leq \texttt{pendack}[\texttt{host}(Pid)])$
$)$
$)$

Proof needs: []

stoller_inv102

Given two processes on the same host where Pid < Pid2, if Pid2 is in the pids-set, then Pid is not alive.

$\forall Pid, Pid2.($
$(((Pid2 > Pid)$
$\wedge\ (Pid2 \in \texttt{pids})$
$\wedge\ (\texttt{host}(Pid) = \texttt{host}(Pid2)))$
$\rightarrow\ (Pid \notin \texttt{alive})$
$)$
$)$

Proof needs: [*stoller_inv2*]

stoller_inv103

If a process (Pid2) is waiting for another process (Pid) and there is an Ack-message from Pid3 (where host(Pid2) = host(Pid3)) in Pid's queue, then Pid2 = Pid3.

$\forall Pid, Pid2, Pid3.($
$(((\texttt{host}(Pid2) \neq \texttt{host}(Pid))$
$\wedge\ (Pid \in \texttt{alive})$
$\wedge\ (Pid2 \in \texttt{alive})$
$\wedge\ \mathbf{elem}(\texttt{m\_Ack}(Pid, Pid3), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge\ (\texttt{host}(Pid2) = \texttt{host}(Pid3))$
$\wedge\ (\texttt{status}[\texttt{host}(Pid2)] = \texttt{wait})$
$\wedge\ (\texttt{host}(\texttt{elid}[\texttt{host}(Pid2)]) = \texttt{host}(Pid)))$
$\rightarrow\ (Pid2 = Pid3)$
$)$
$)$

Proof needs: [*stoller_inv2*, *stoller_inv9*, *stoller_inv14*, *stoller_inv78*, *stoller_inv102*]

stoller_inv104

There are never two Ack-messages in one queue, which acknowledges the same process (Pid), from two different process (Pid2 and Pid3) on the same host.

$\forall Pid, Pid2, Pid3.($
$(((Pid2 \neq Pid3)$
$\wedge\ \mathbf{elem}(\texttt{m\_Ack}(Pid, Pid2), \texttt{queue}(\texttt{host}(Pid)))$
$\wedge\ (\texttt{host}(Pid2) = \texttt{host}(Pid3)))$
$\rightarrow\ \neg\mathbf{elem}(\texttt{m\_Ack}(Pid, Pid3), \texttt{queue}(\texttt{host}(Pid)))$
$)$
$)$

Proof needs: [*stoller_inv3*, *stoller_inv14*, *stoller_inv15*]